# A Scalable Architecture for Ordered Parallelism

**Mark Jeffrey**, Suvinay Subramanian, Cong Yan, Joel Emer, Daniel Sanchez

MICRO 2015

Massachusetts Institute of Technology

CSAIL

NVIDIA

# Multicores Target Easy Parallelism

# Multicores Target Easy Parallelism

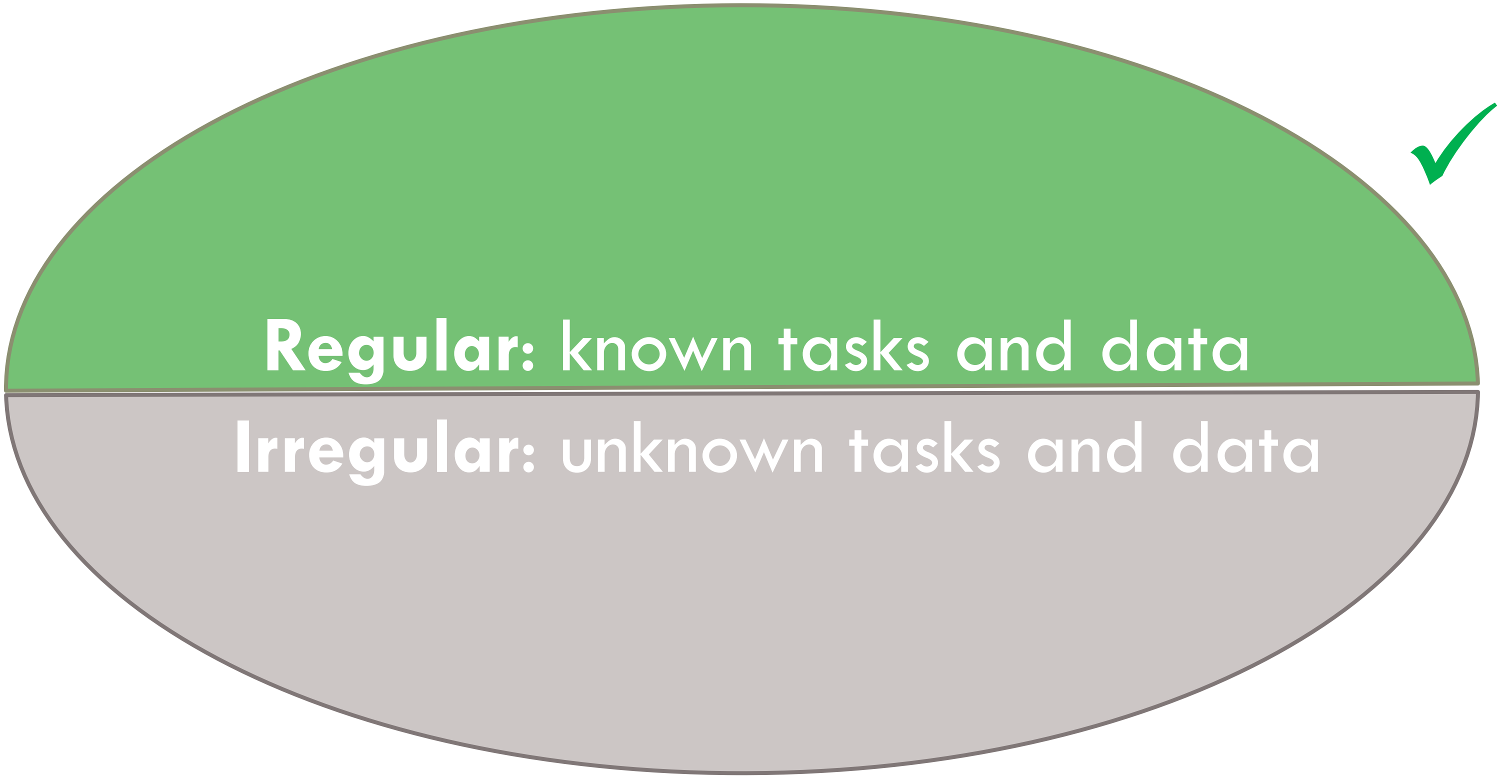
**Regular:** known tasks and data

# Multicores Target Easy Parallelism

**Regular:** known tasks and data

✓

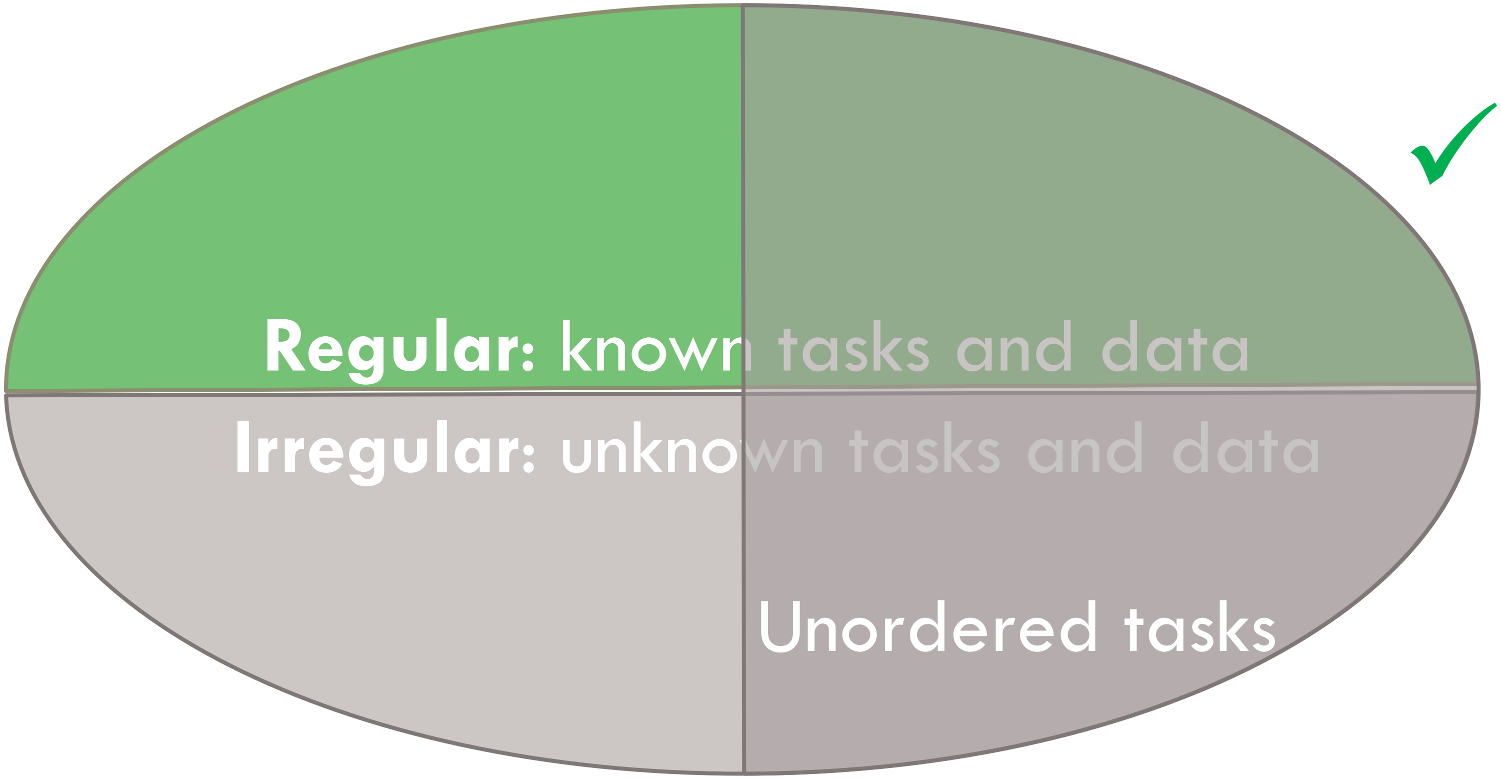
# Multicores Target Easy Parallelism



**Regular:** known tasks and data ✓

**Irregular:** unknown tasks and data

# Multicores Target Easy Parallelism

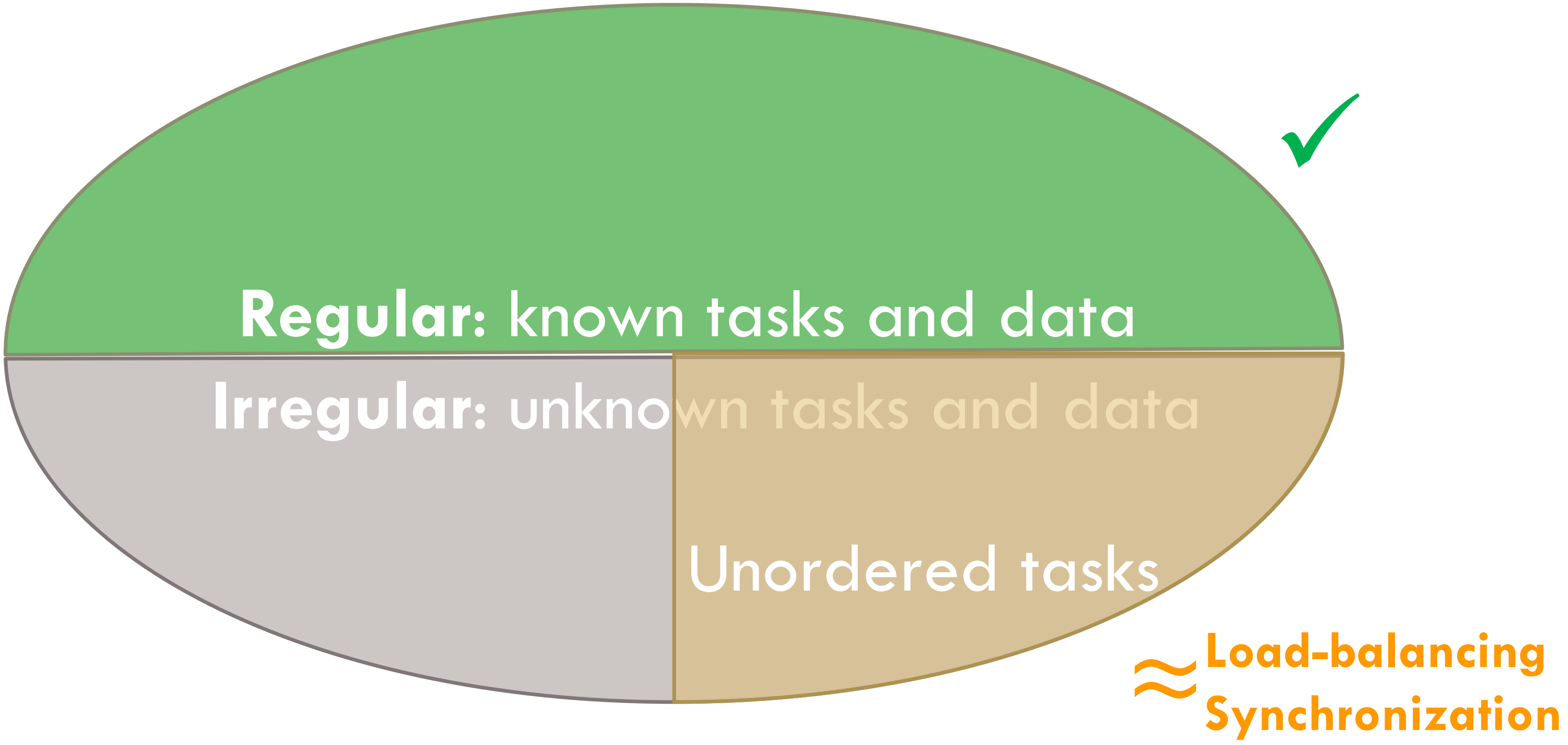


**Regular:** known tasks and data

**Irregular:** unknown tasks and data

Unordered tasks

# Multicores Target Easy Parallelism



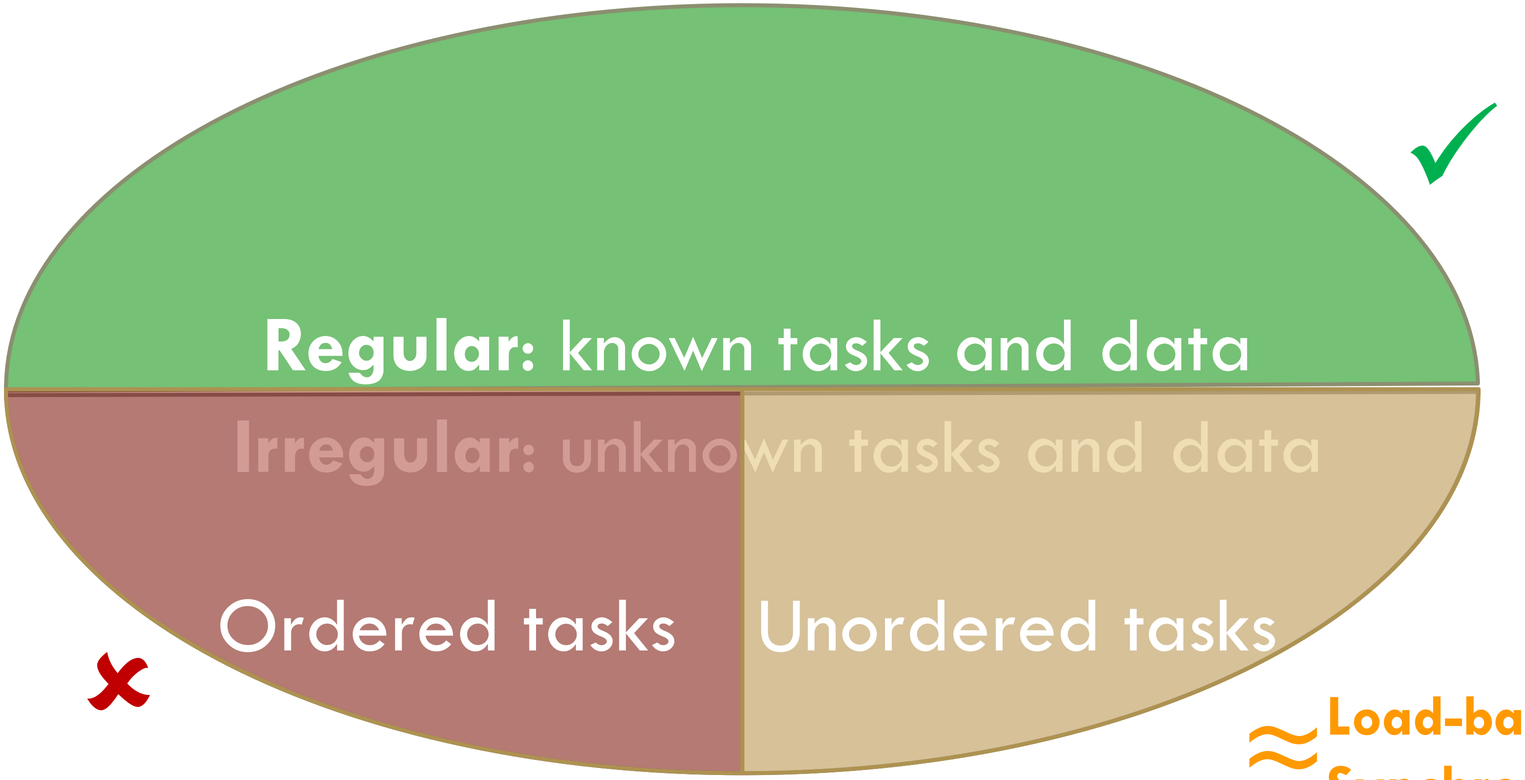
**Regular:** known tasks and data ✓

**Irregular:** unknown tasks and data

Unordered tasks

≈ **Load-balancing**
**Synchronization**

# Multicores Target Easy Parallelism



**Regular:** known tasks and data ✓

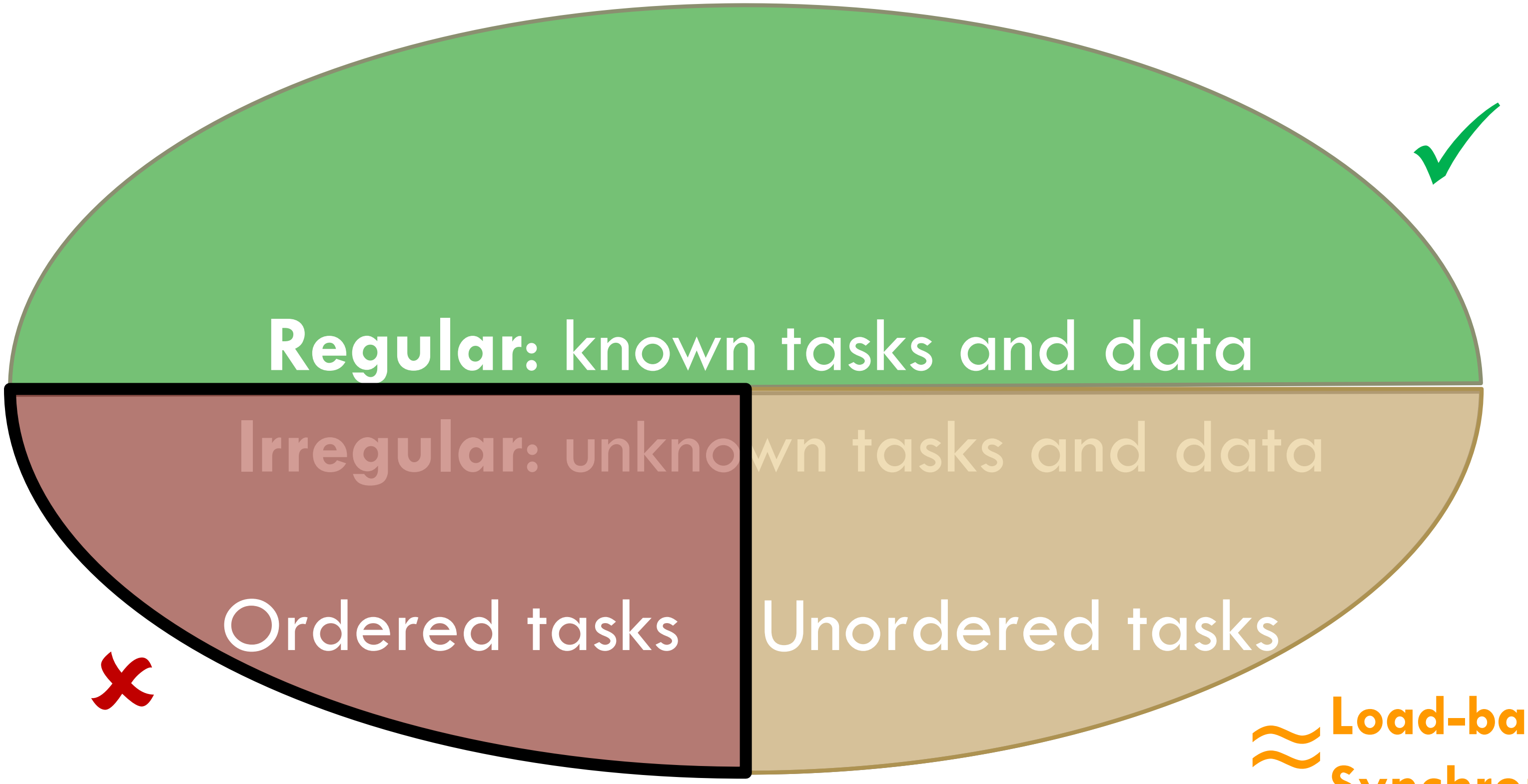
**Irregular:** unknown tasks and data

Ordered tasks ✗

Unordered tasks

≈ **Load-balancing**
**Synchronization**

# Multicores Target Easy Parallelism



✓

**Regular:** known tasks and data

**Irregular:** unknown tasks and data

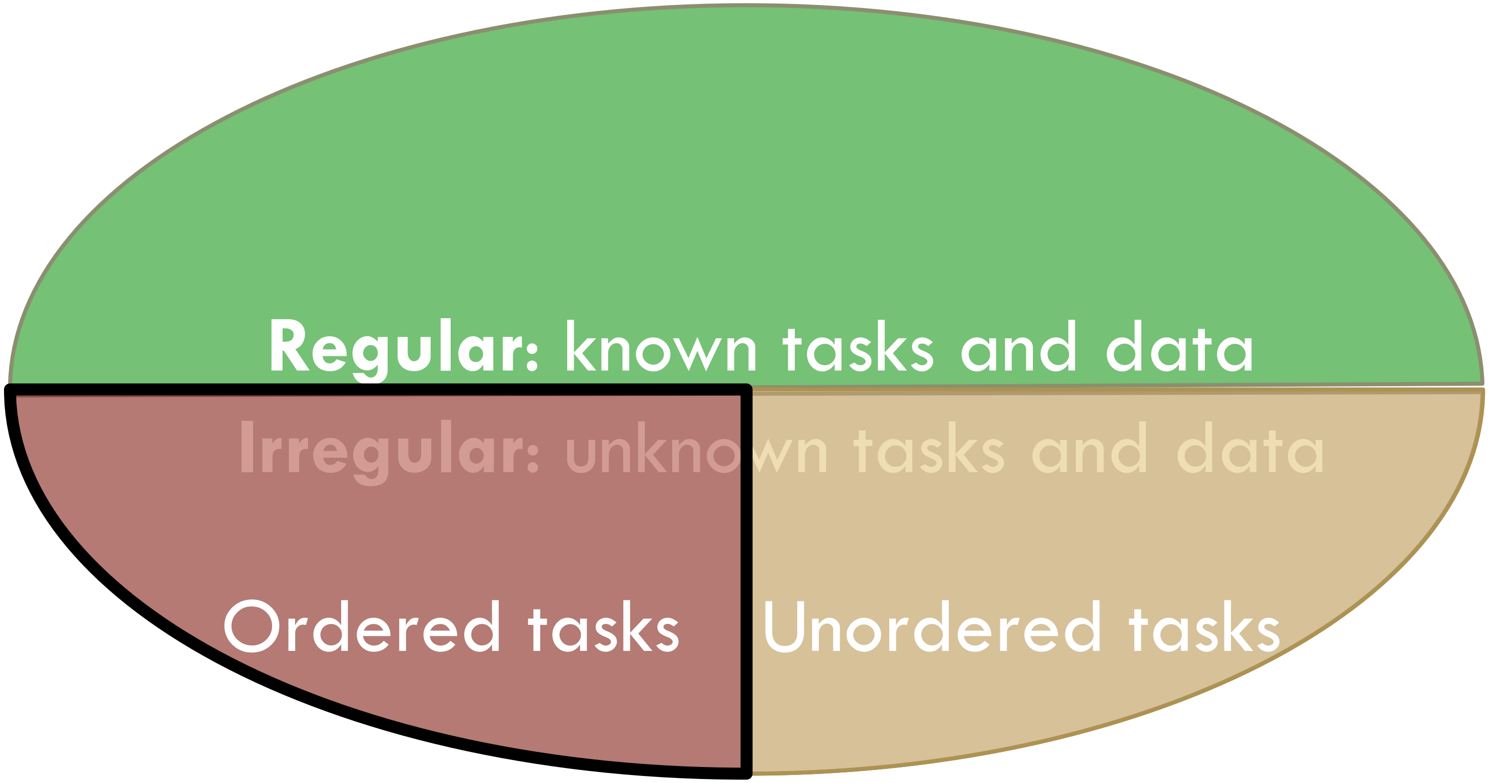
Ordered tasks

Unordered tasks

✗

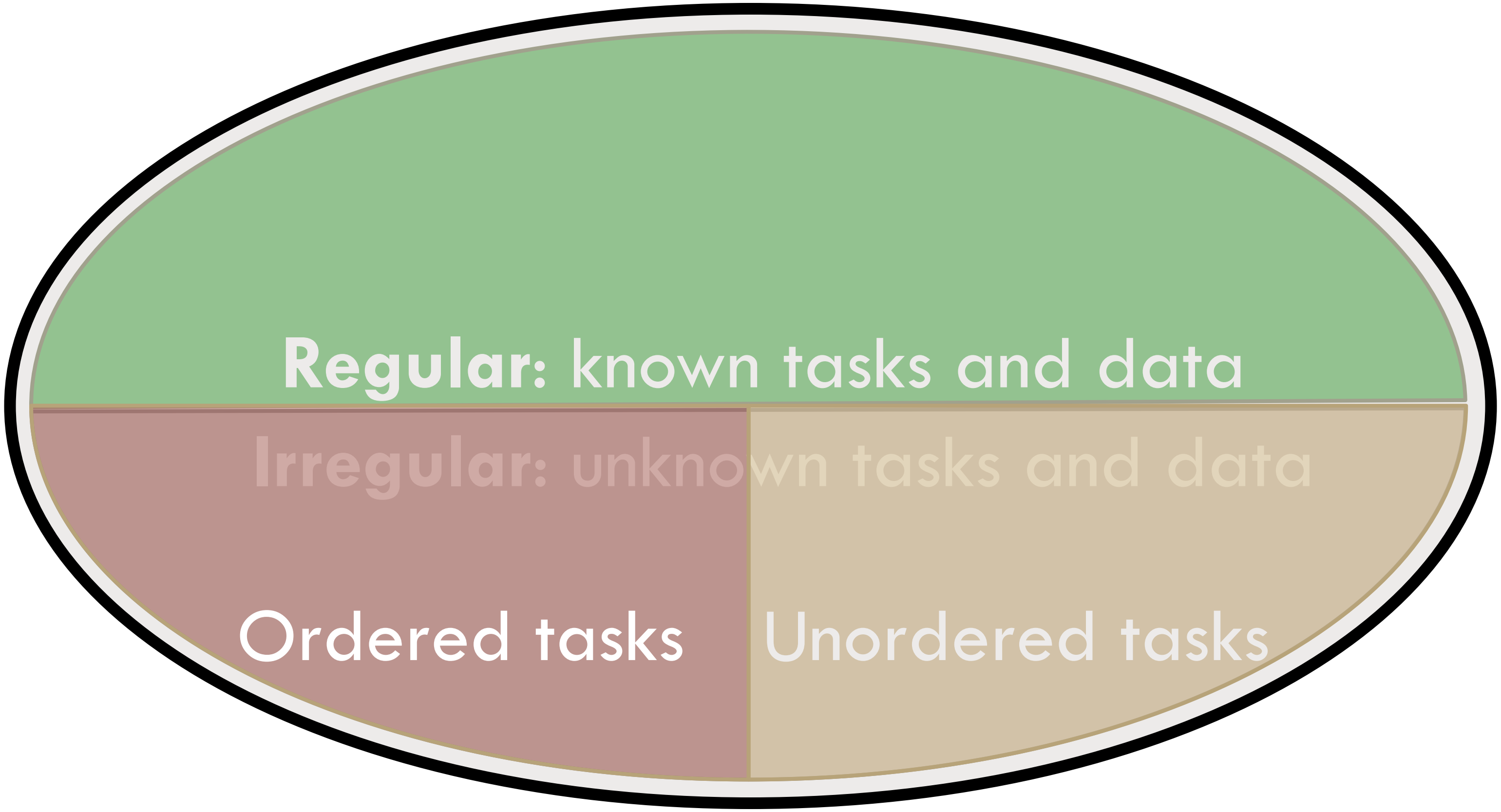
≈ **Load-balancing**
**Synchronization**

# Multicores Target Easy Parallelism


**Regular:** known tasks and data
**Irregular:** unknown tasks and data
Ordered tasks
Unordered tasks

Ordering is a simple and general form of synchronization

# Multicores Target Easy Parallelism


**Regular:** known tasks and data
**Irregular:** unknown tasks and data
Ordered tasks
Unordered tasks

Ordering is a simple and general form of synchronization

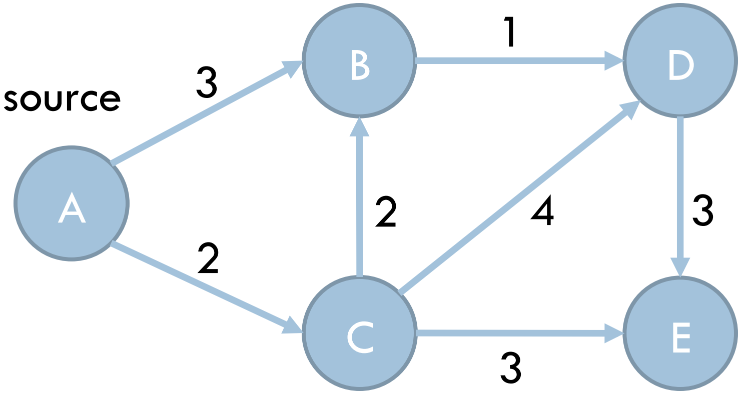
Support for **order** enables widespread parallelism

# Outline

- □ Understanding Ordered Parallelism
- □ Swarm
- □ Evaluation

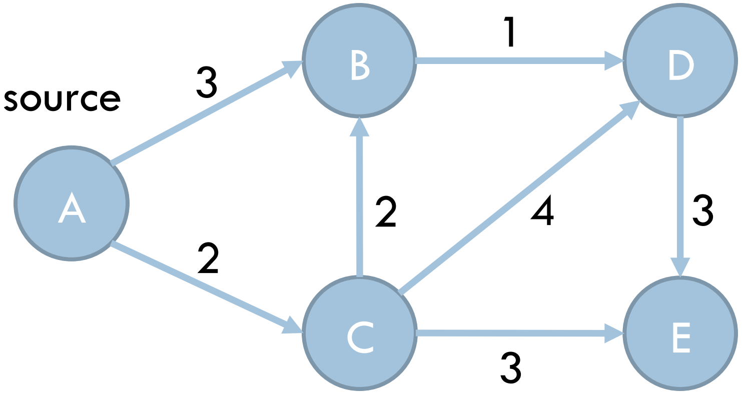
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges

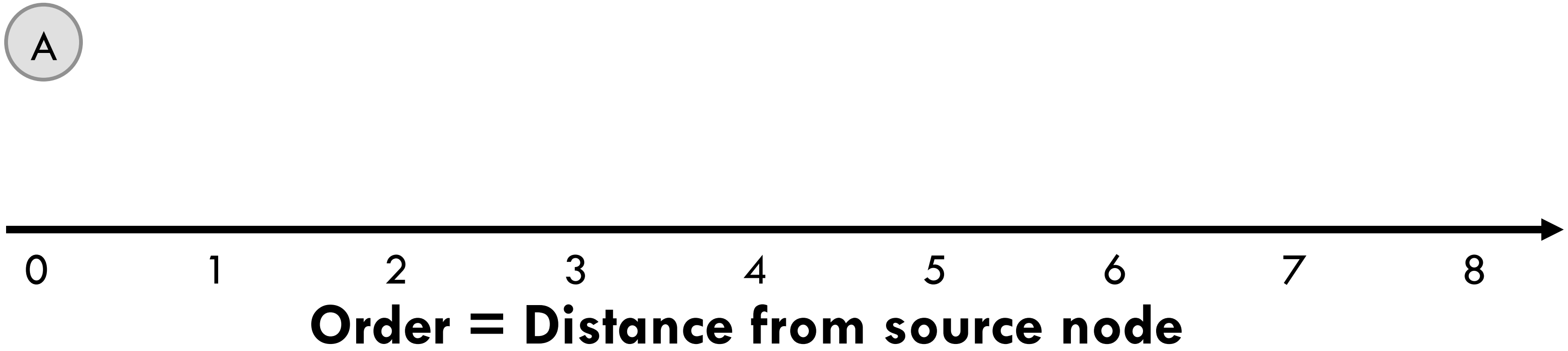


# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges

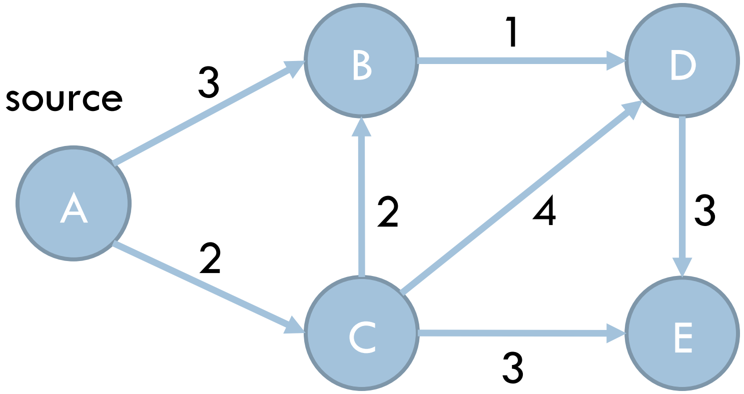


**Tasks**

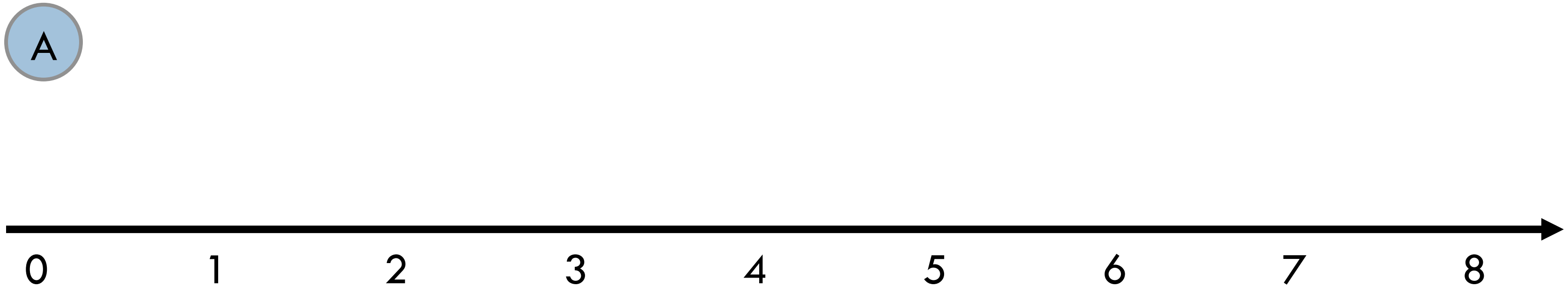


# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges



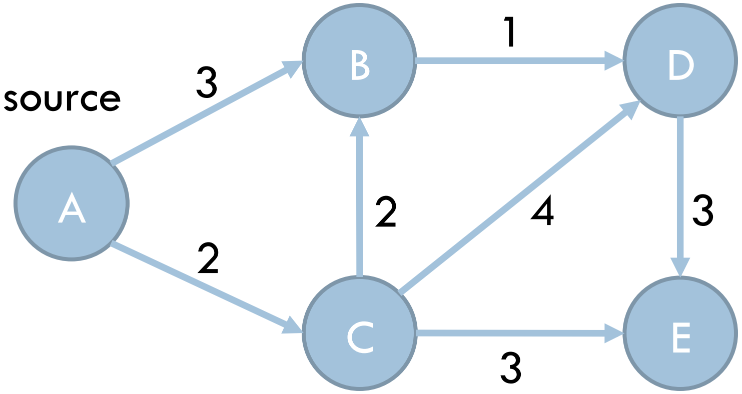
**Tasks**

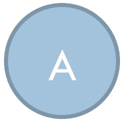


**Order = Distance from source node**

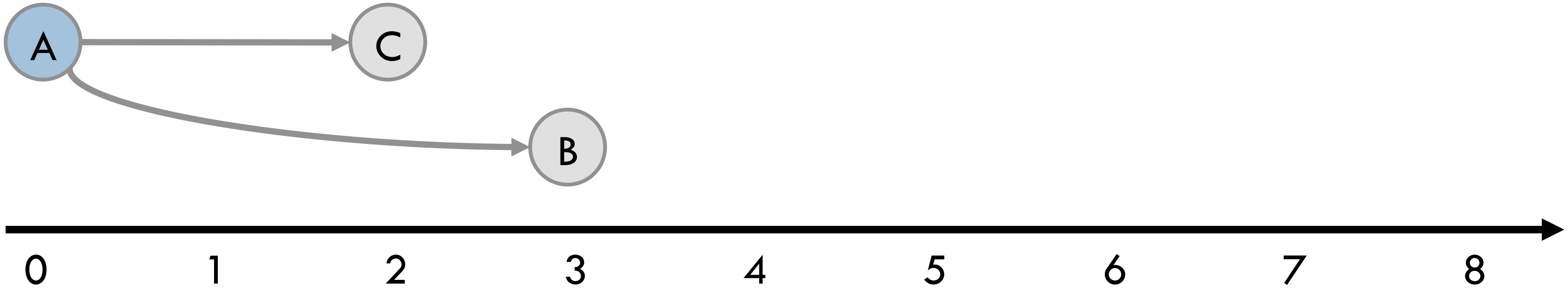
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





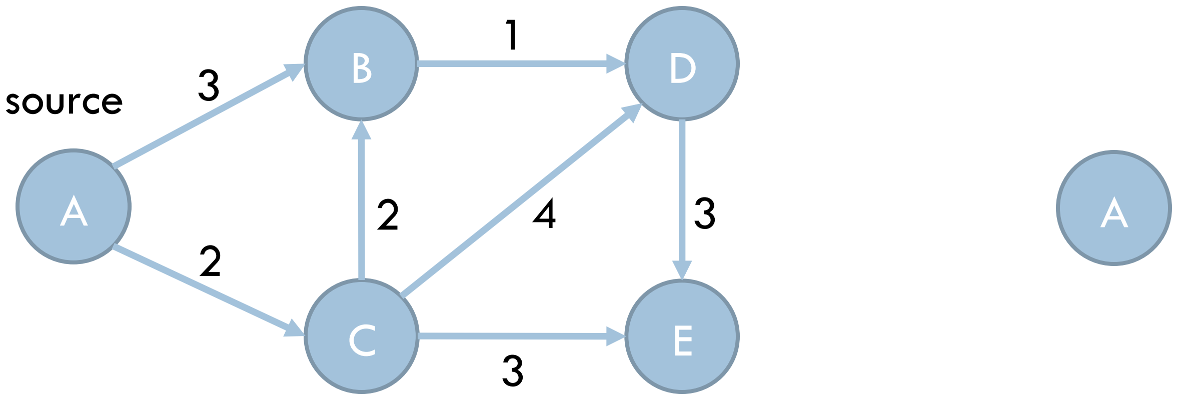
**Tasks**



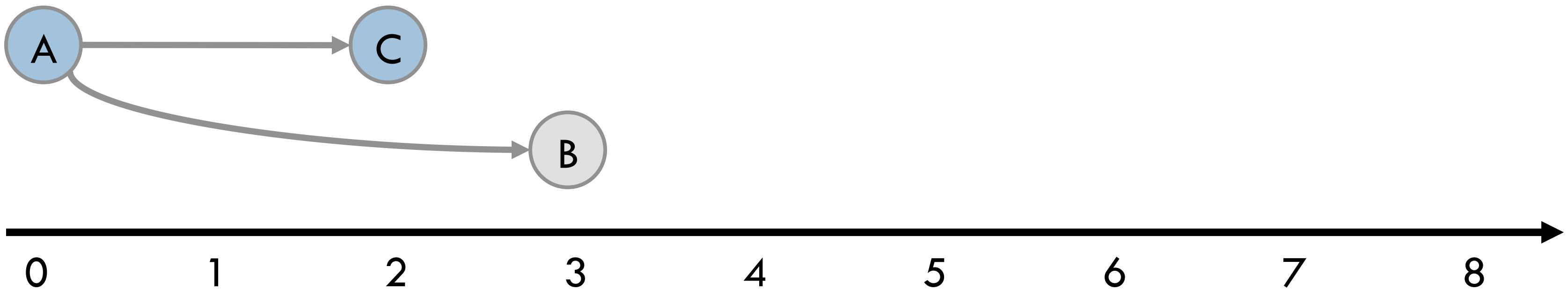
**Order = Distance from source node**

# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges



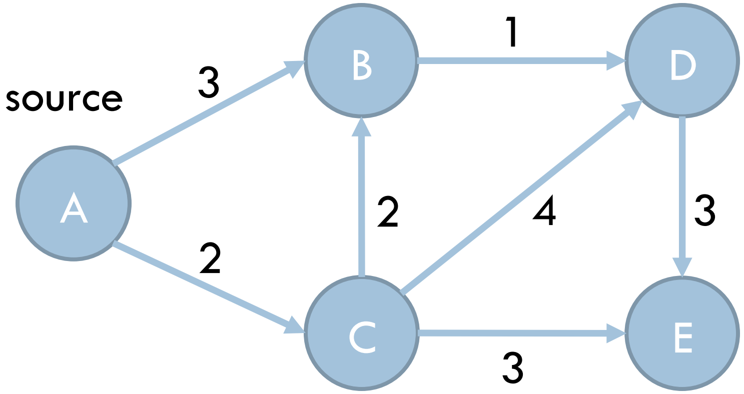
**Tasks**

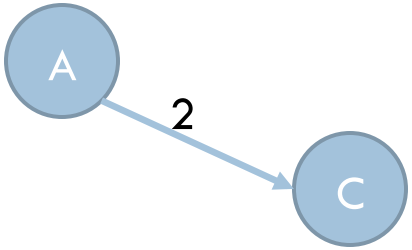


**Order = Distance from source node**

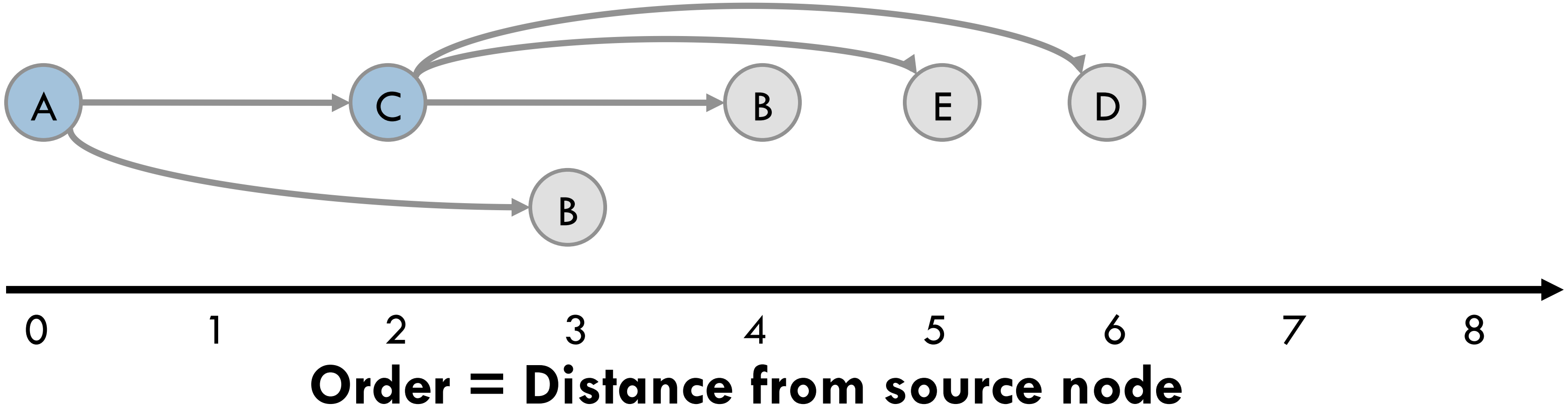
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges



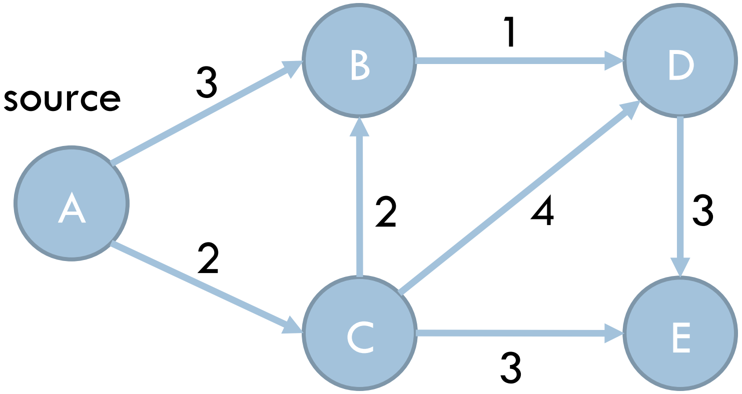


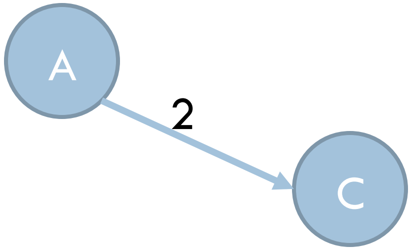
**Tasks**



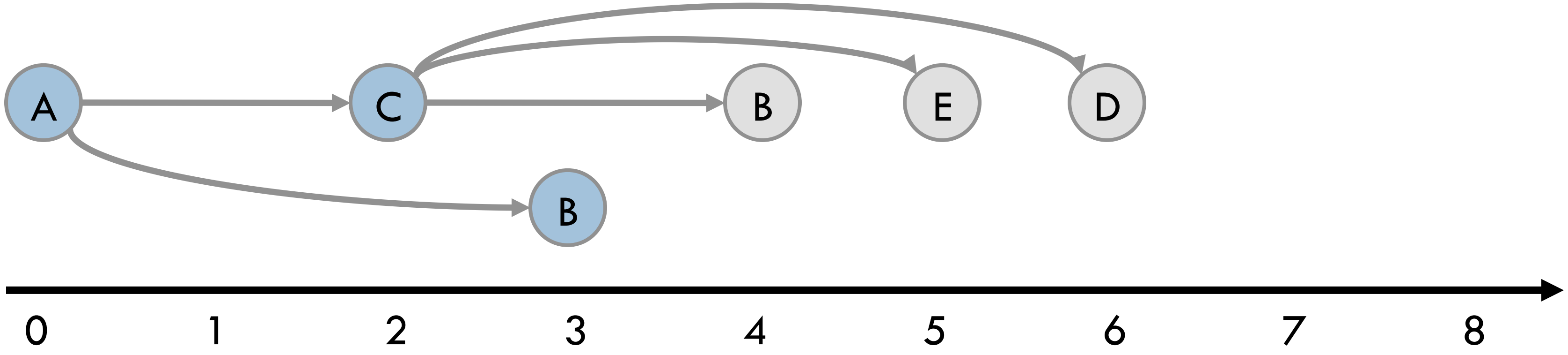
# Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges





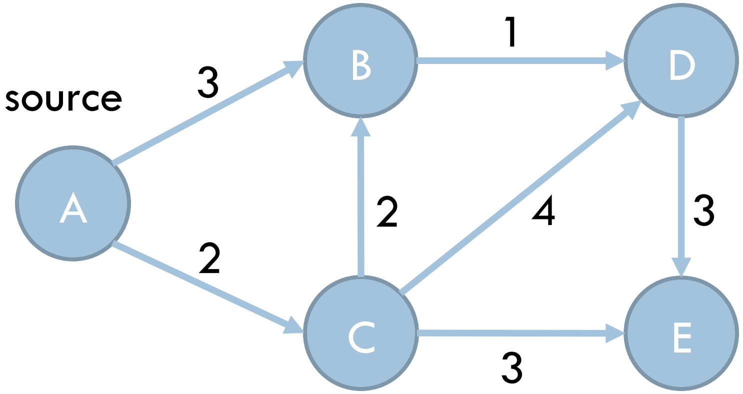
**Tasks**

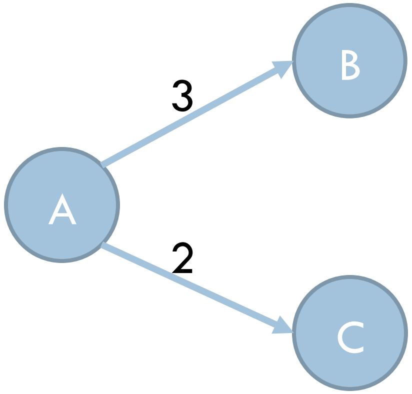


**Order = Distance from source node**

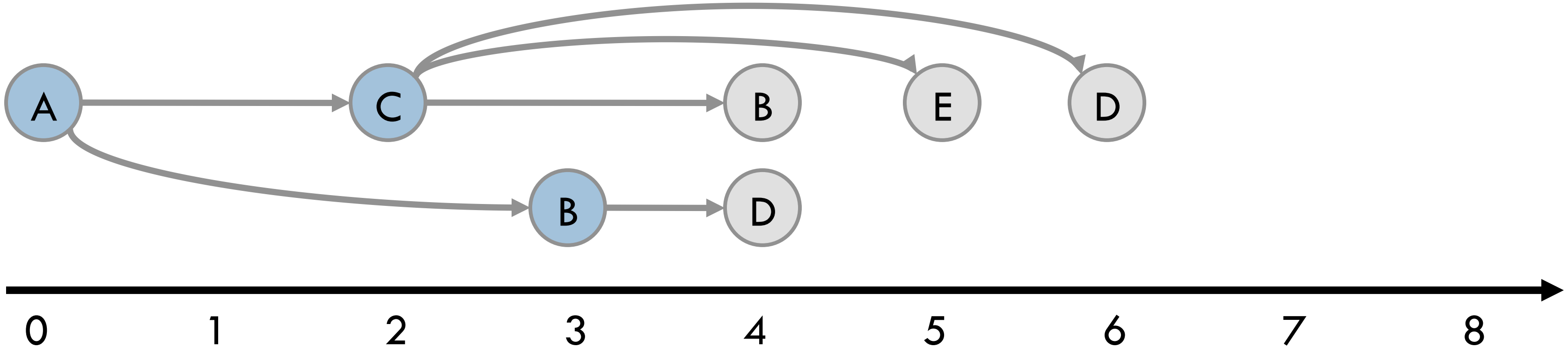
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





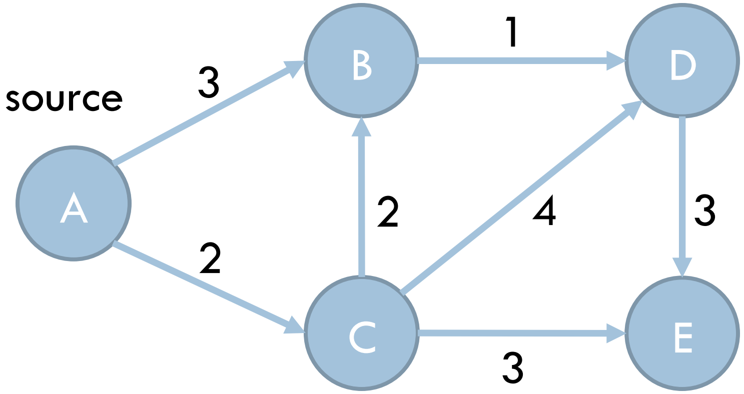
**Tasks**

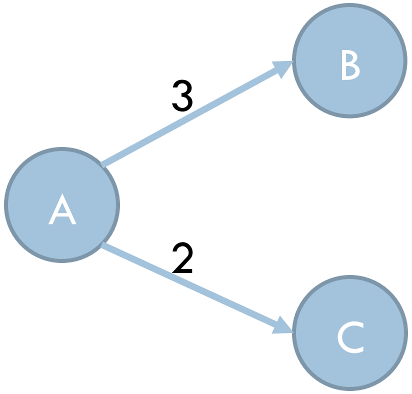


**Order = Distance from source node**

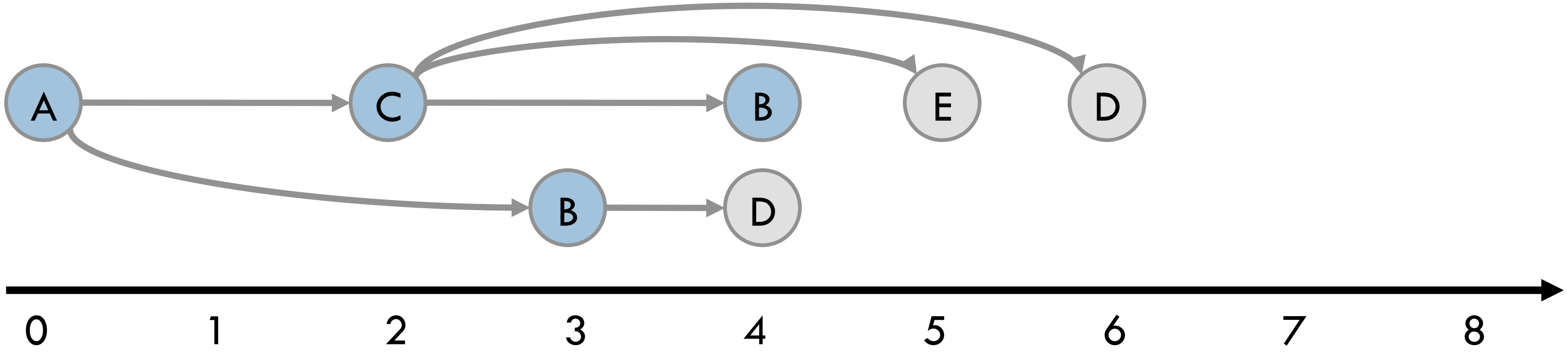
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





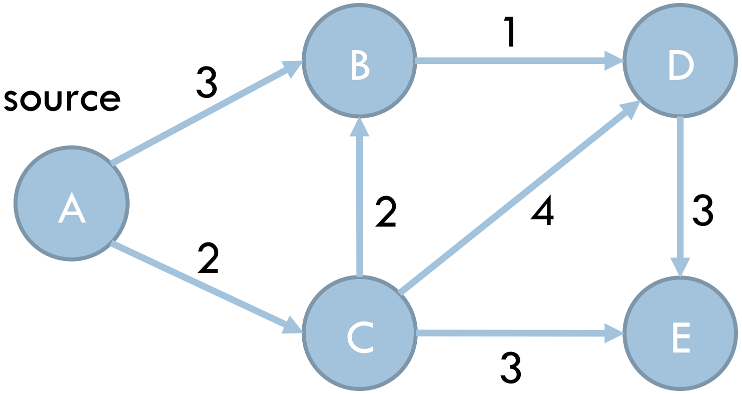
**Tasks**

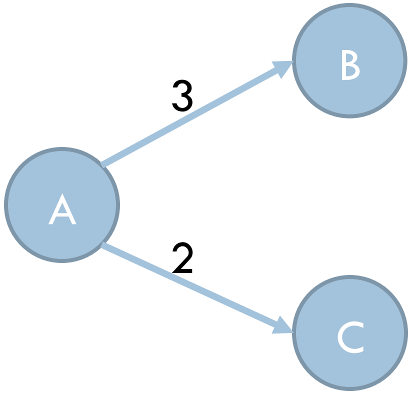


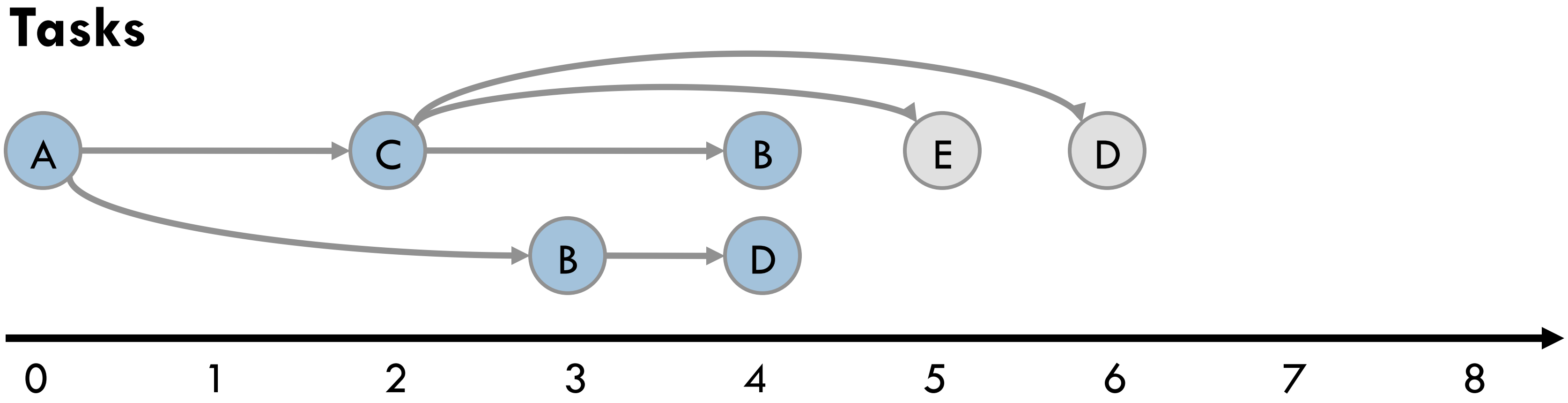
**Order = Distance from source node**

# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges



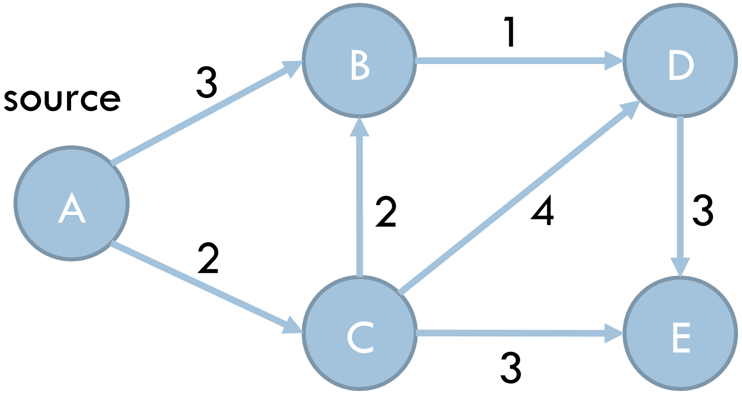


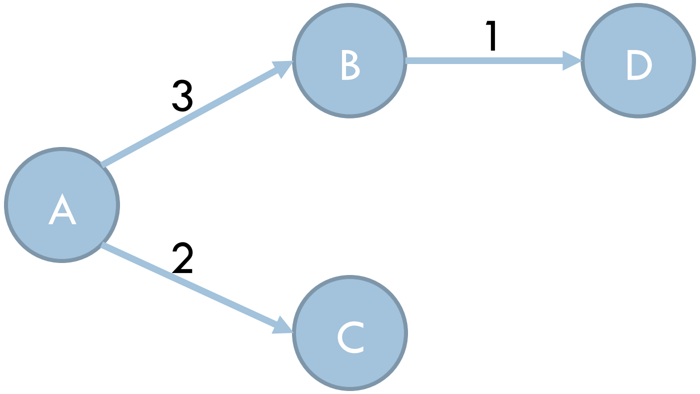


**Order = Distance from source node**

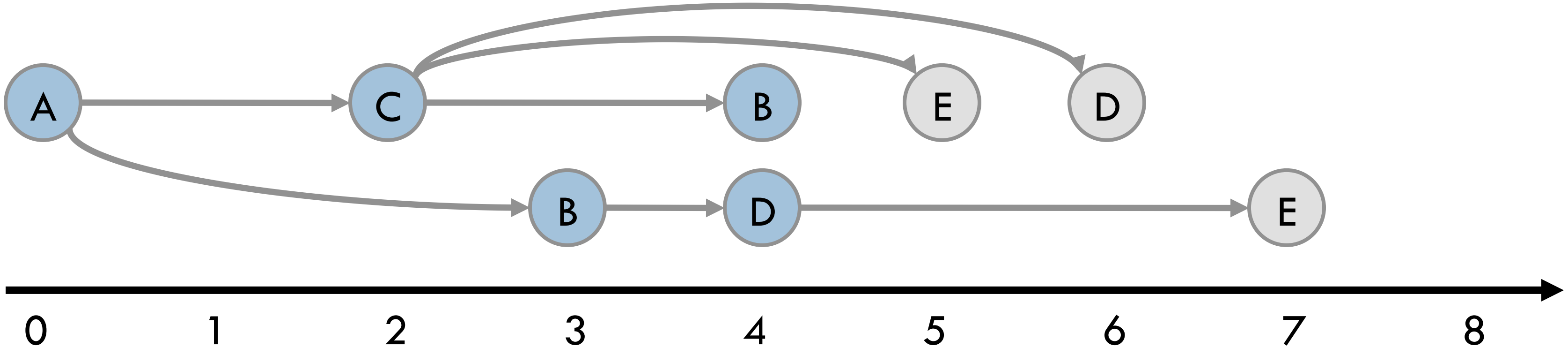
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





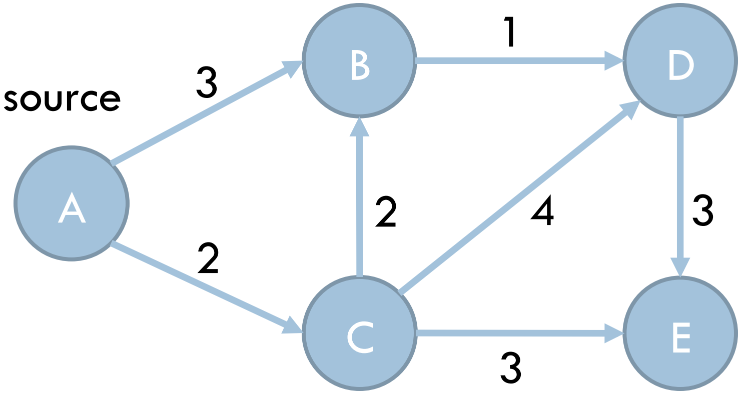
**Tasks**

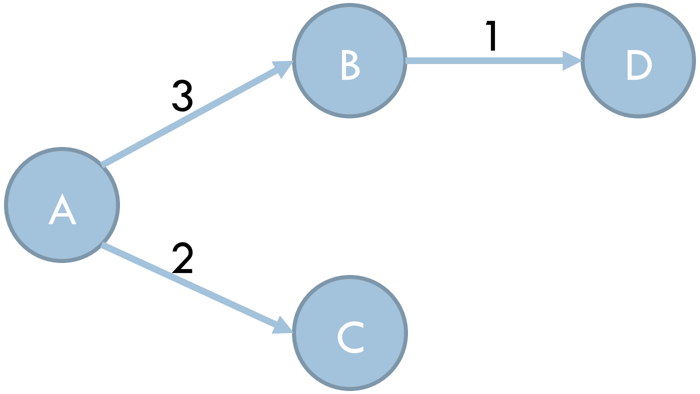


**Order = Distance from source node**

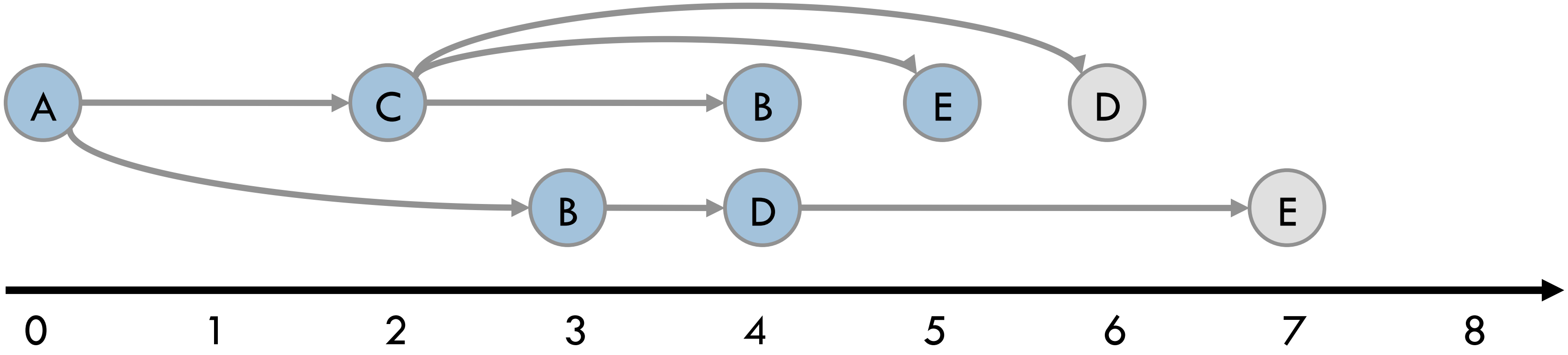
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





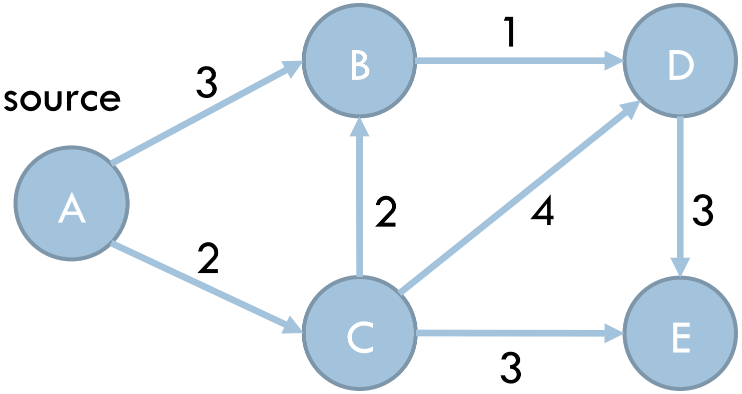
**Tasks**

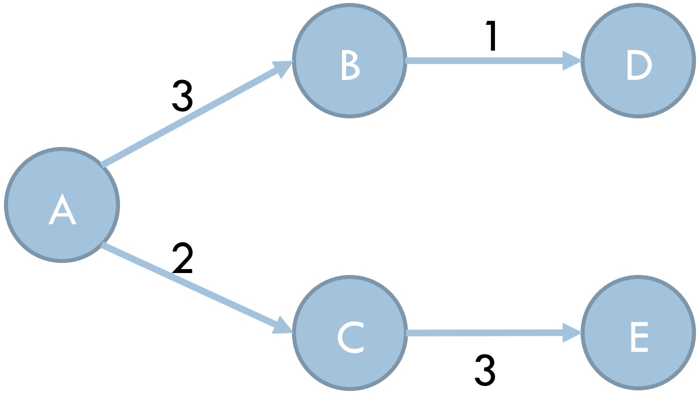


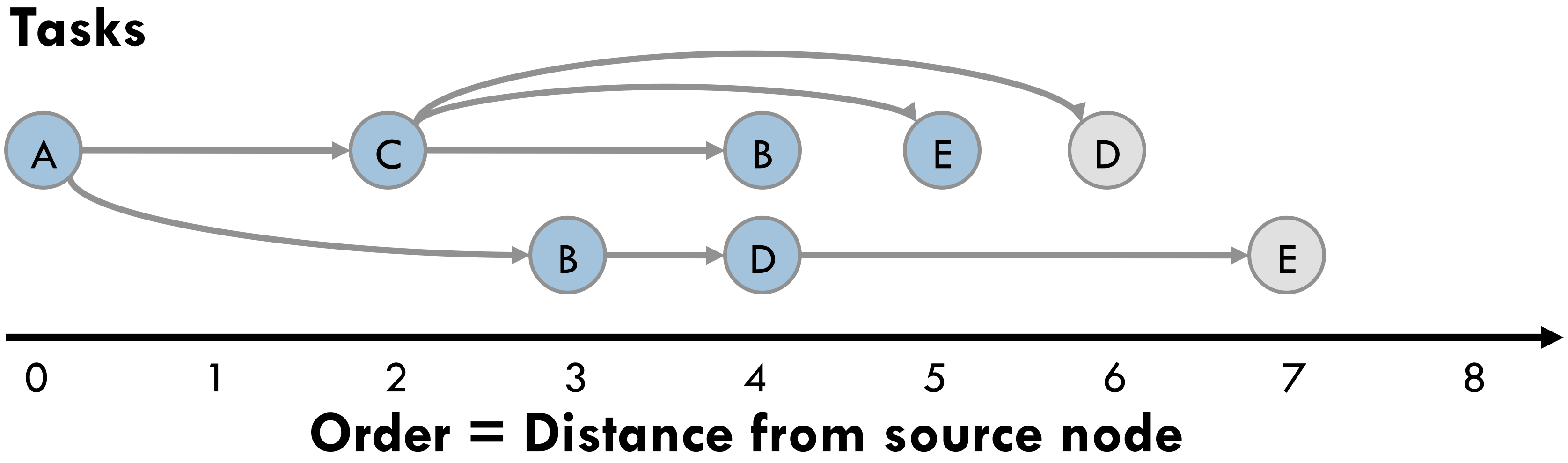
**Order = Distance from source node**

# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges

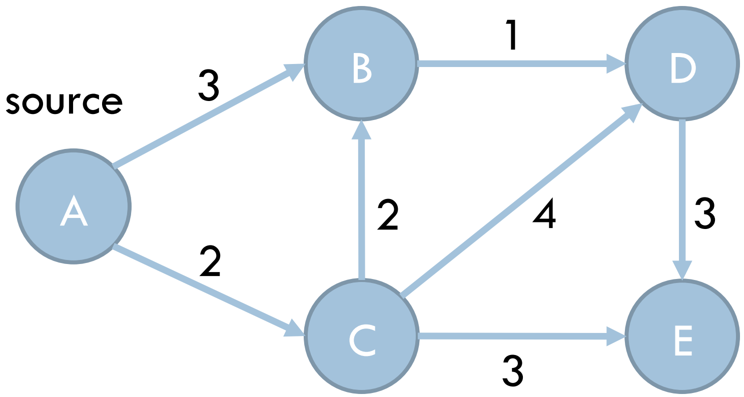


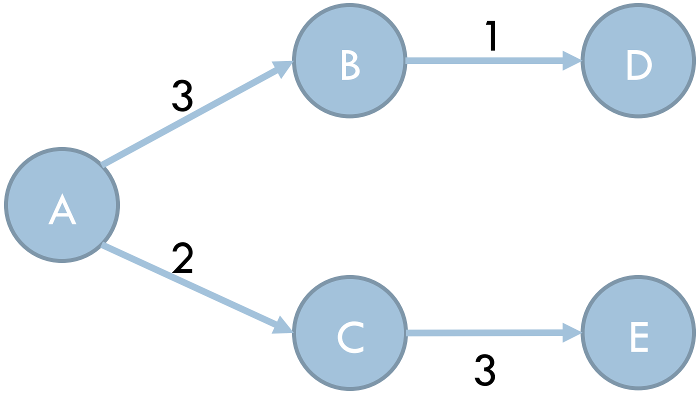




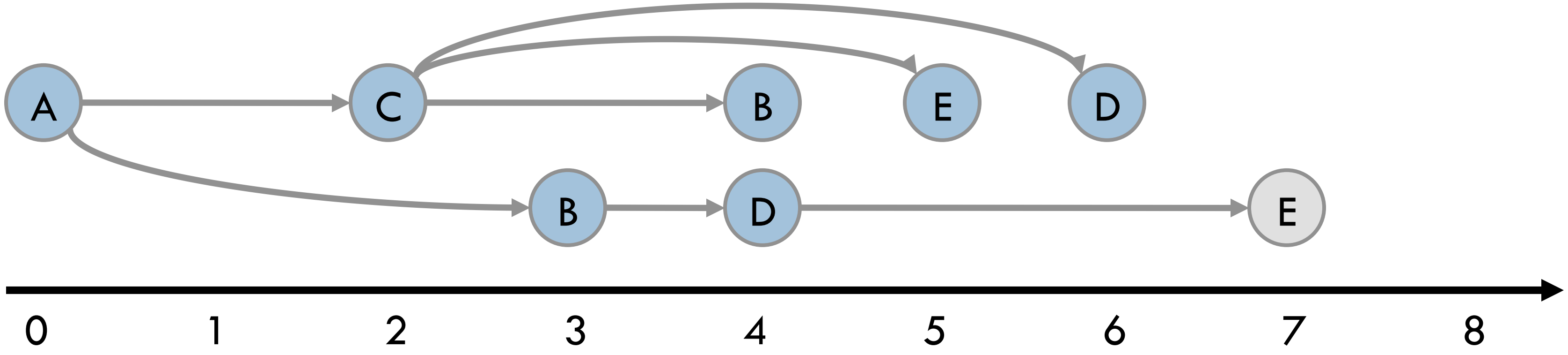
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





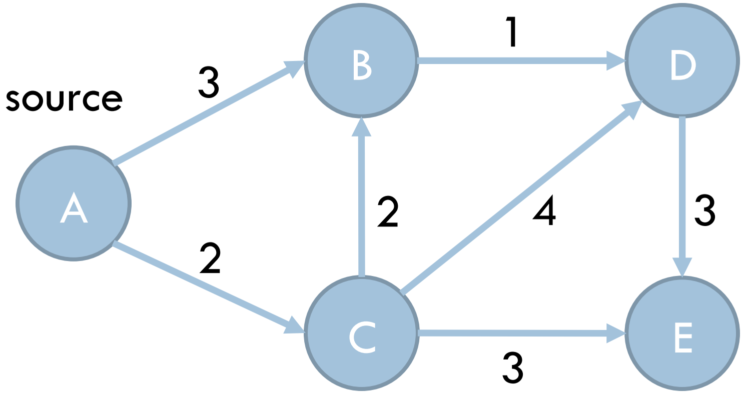
**Tasks**

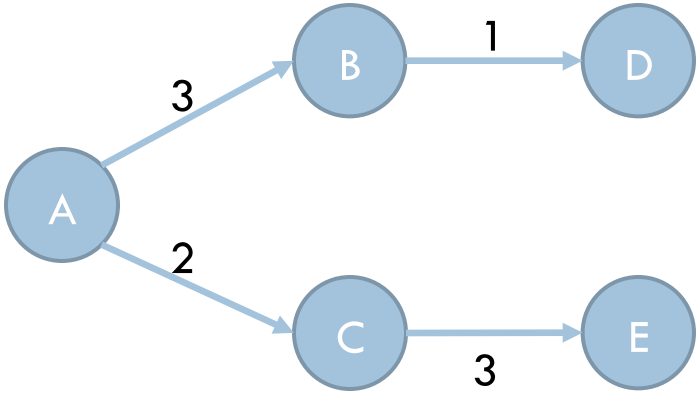


**Order = Distance from source node**

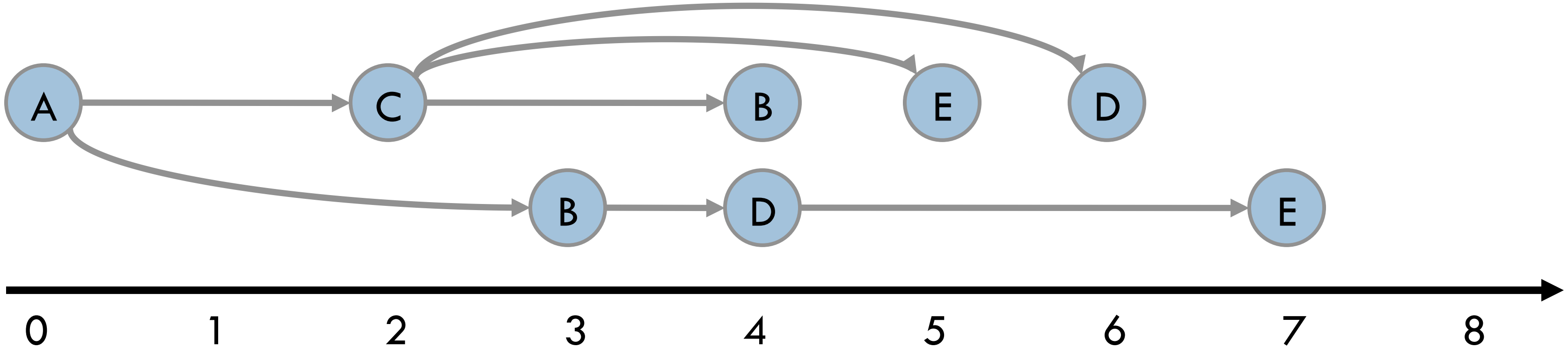
# Example: Parallelism in Dijkstra's Algorithm

Finds shortest-path tree on a graph with weighted edges





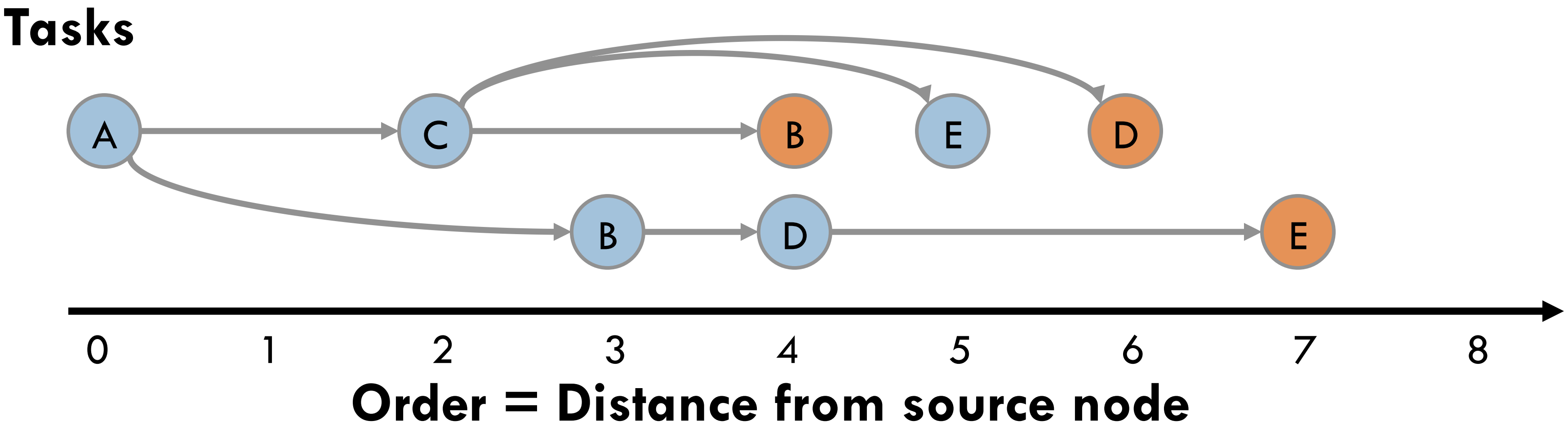
**Tasks**



**Order = Distance from source node**

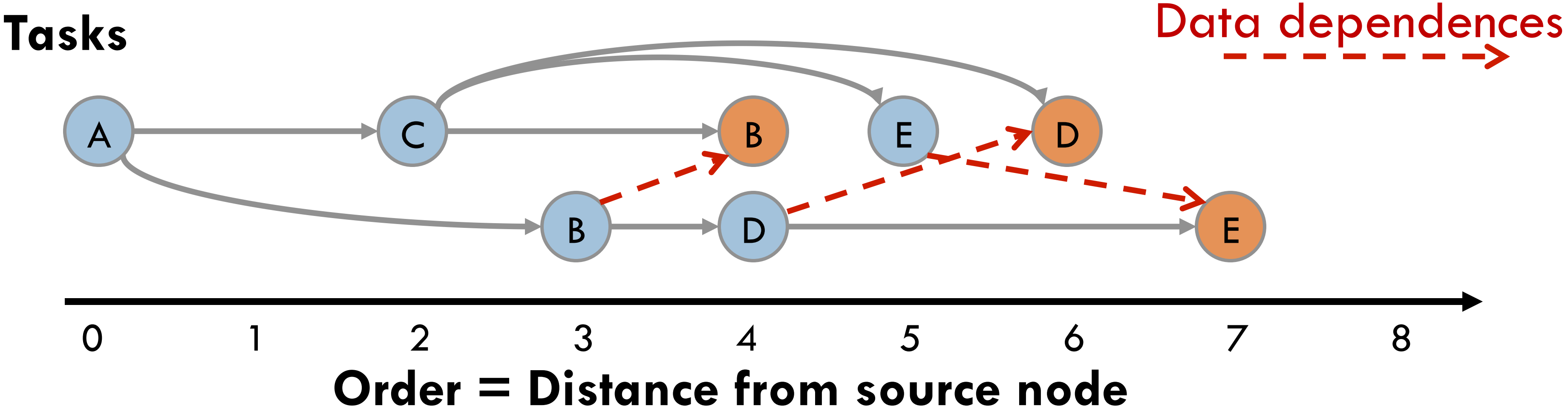
# Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order



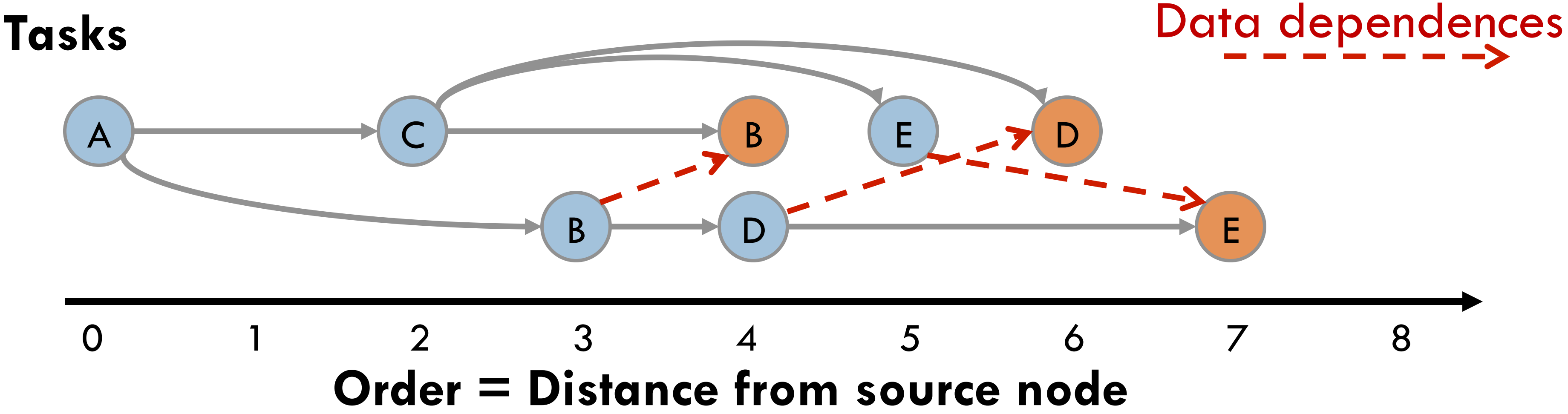
# Parallelism in Dijkstra's Algorithm

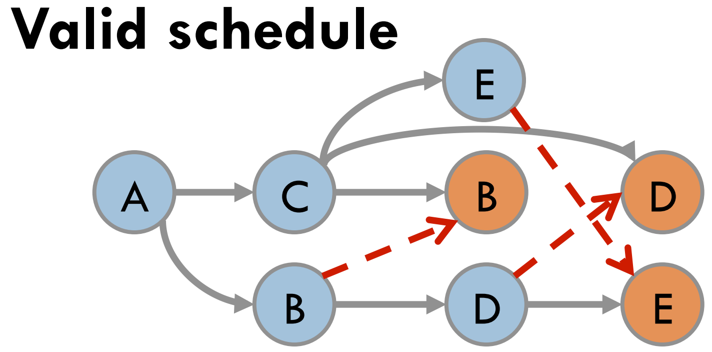
Can execute independent tasks out of order



# Parallelism in Dijkstra's Algorithm

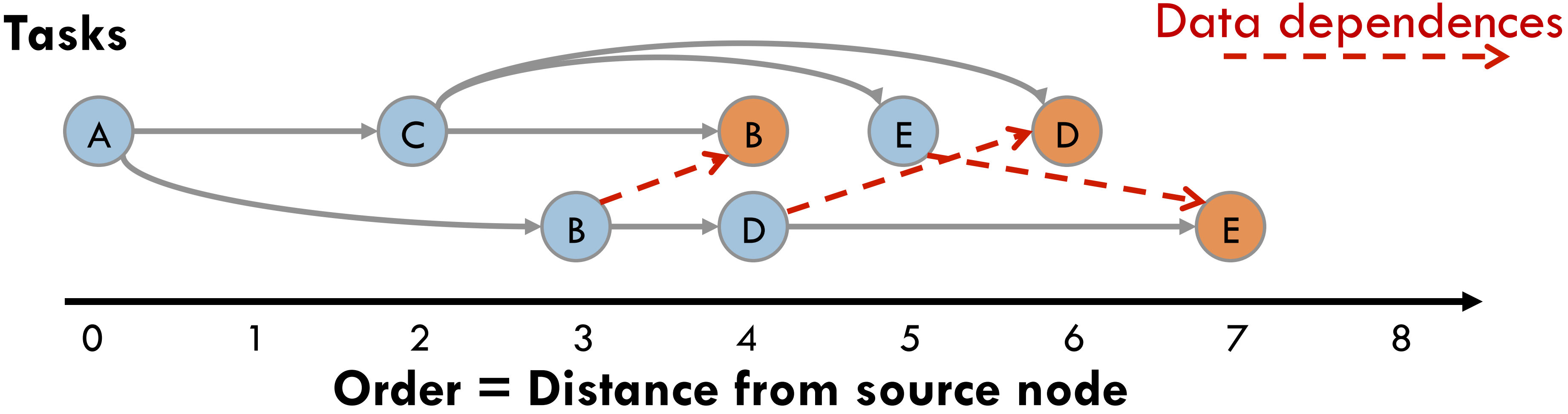
Can execute independent tasks out of order

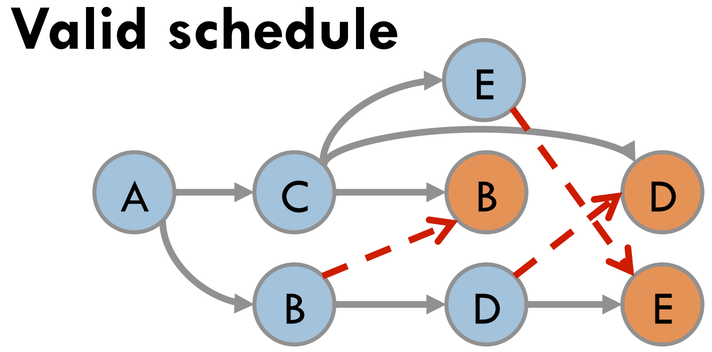




# Parallelism in Dijkstra's Algorithm

Can execute independent tasks out of order





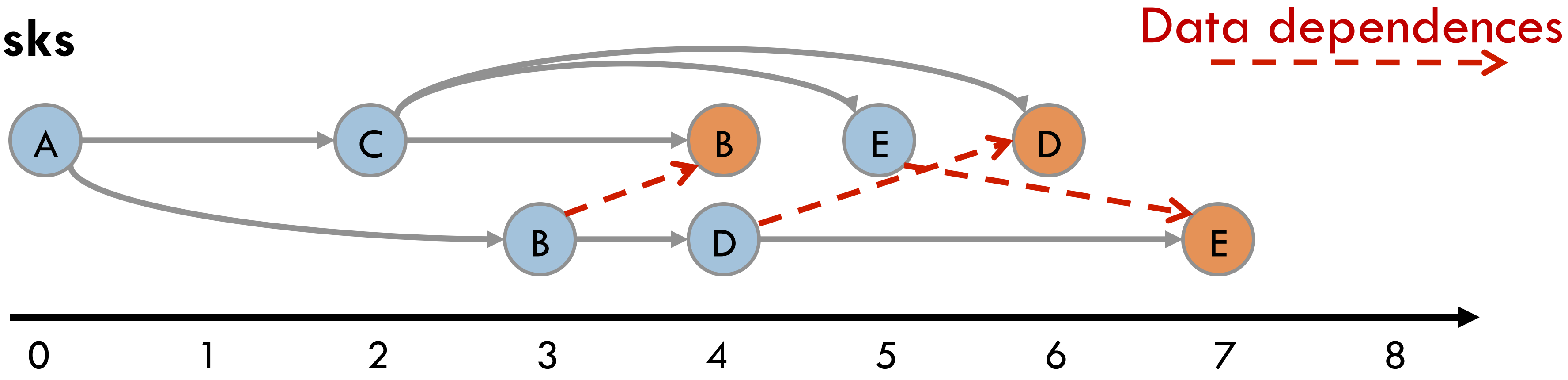
2x parallelism (more in larger graphs)

Tasks and dependences unknown in advance

# Parallelism in Dijkstra's Algorithm

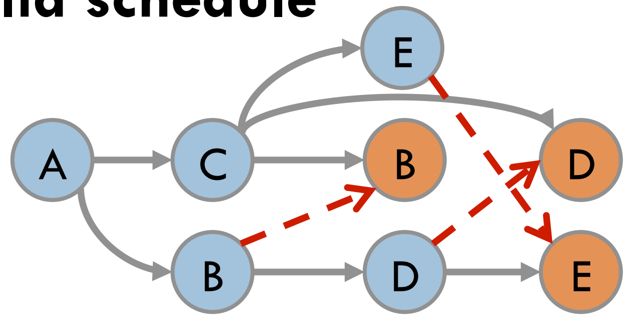
Can execute independent tasks out of order

**Tasks**



**Order = Distance from source node**

**Valid schedule**



**2x parallelism**
**(more in larger graphs)**

**Tasks and dependences unknown in advance**

Need speculative execution to elide order constraints

# Insights about Ordered Parallelism

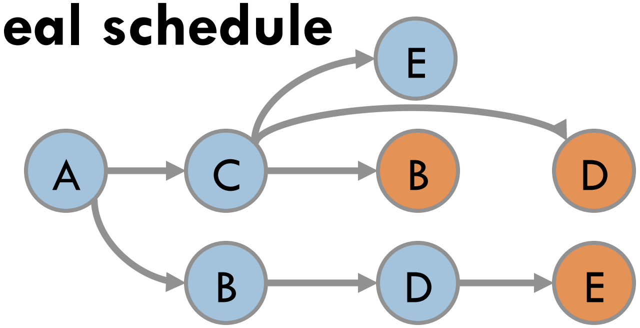
# Insights about Ordered Parallelism

1. **With perfect speculation, parallelism is plentiful**

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

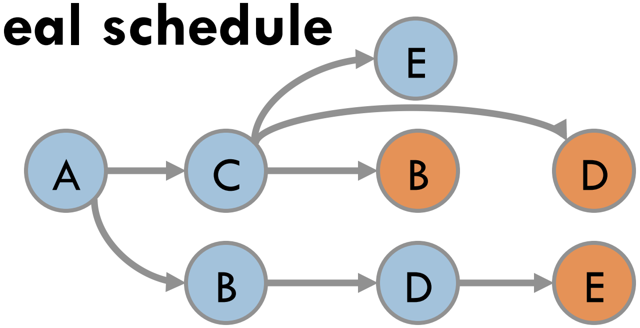
**Ideal schedule**



# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

**Ideal schedule**



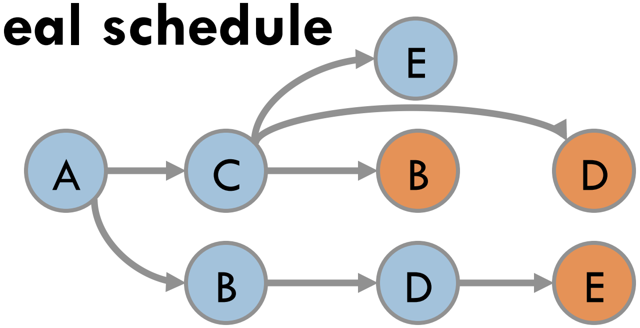
Parallelism

| | |
|---|---|
| max | 800x |

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

**Ideal schedule**



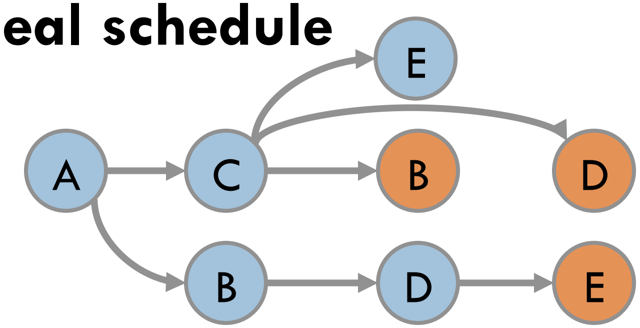
| | Parallelism |
|---|---|
| max | 800x |

## 2. Tasks are tiny: 32 instructions on average

# Insights about Ordered Parallelism

1. **With perfect speculation, parallelism is plentiful**

**Ideal schedule**



Parallelism

| | |
|---|---|
| max | 800x |

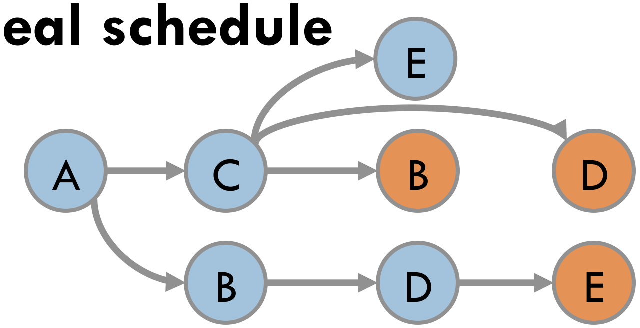
2. **Tasks are tiny:** 32 instructions on average

3. **Independent tasks are far away in program order**

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

**Ideal schedule**

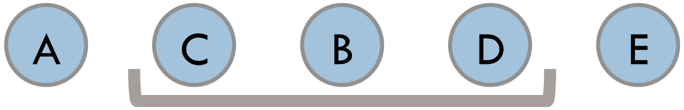


Parallelism

| | |
|---|---|
| max | 800x |

## 2. Tasks are tiny: 32 instructions on average

## 3. Independent tasks are far away in program order



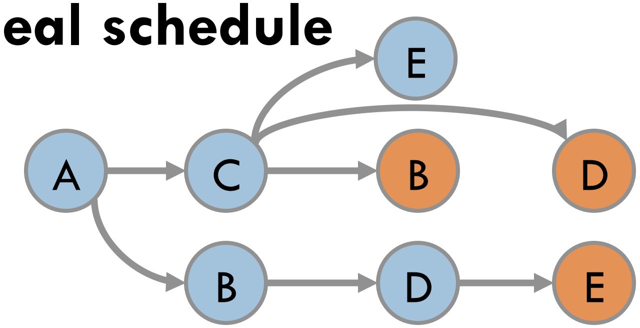
*N*-task window

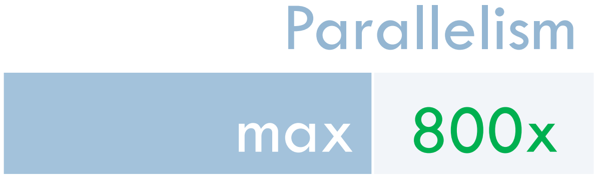
Can execute *N* tasks ahead of the earliest active task

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

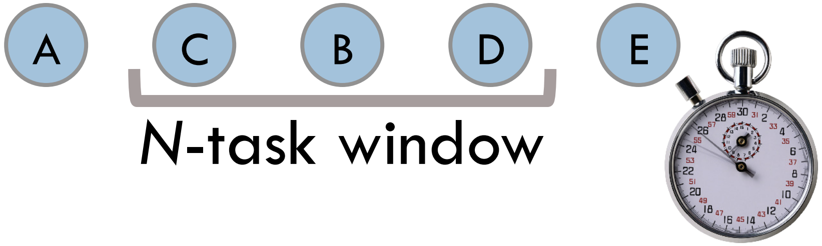
**Ideal schedule**





## 2. Tasks are tiny: 32 instructions on average

## 3. Independent tasks are far away in program order

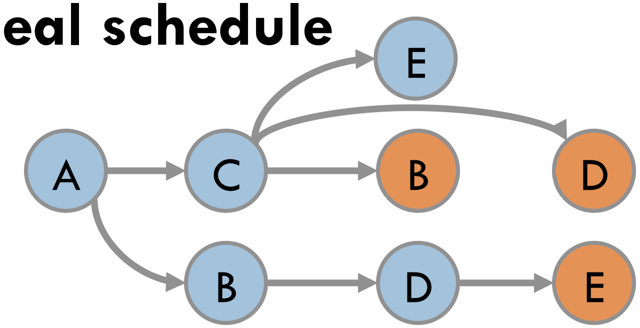


Can execute *N* tasks ahead of the earliest active task

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

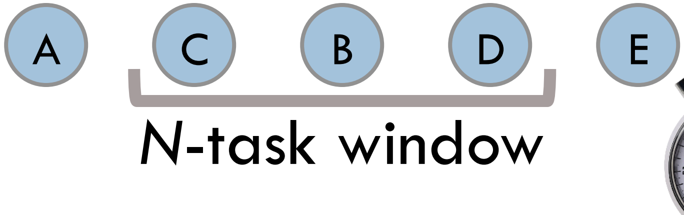
**Ideal schedule**



| | Parallelism |
|---|---|
| max | 800x |
| window=64 | 26x |

## 2. Tasks are tiny: 32 instructions on average

## 3. Independent tasks are far away in program order

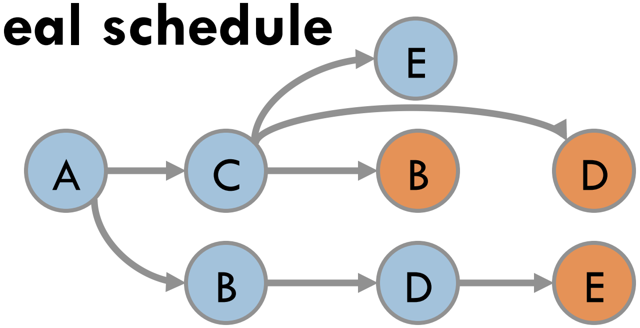


Can execute *N* tasks ahead of the earliest active task

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

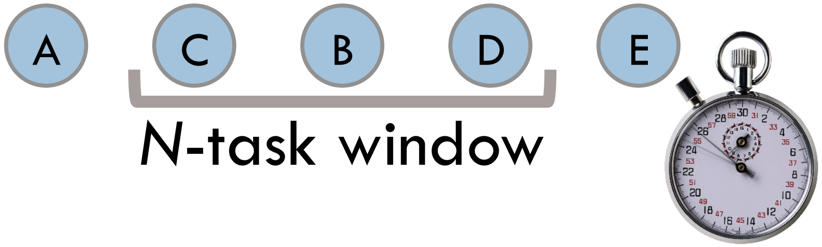
**Ideal schedule**



| | Parallelism |
|---|---|
| max | 800x |
| window=64 | 26x |
| window=1k | 180x |

## 2. Tasks are tiny: 32 instructions on average

## 3. Independent tasks are far away in program order



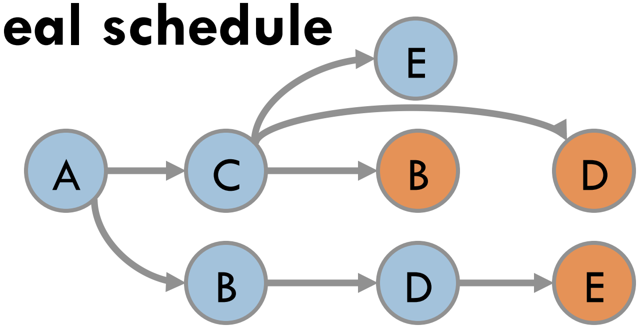
*N*-task window

Can execute *N* tasks ahead of the earliest active task

# Insights about Ordered Parallelism

## 1. With perfect speculation, parallelism is plentiful

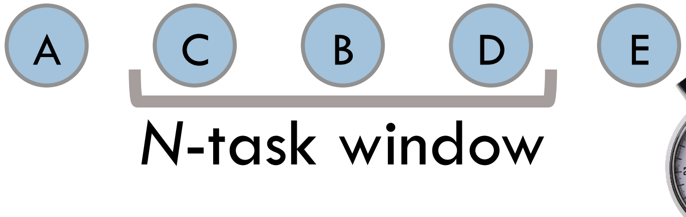
**Ideal schedule**



| Parallelism | |
|---|---|
| max | 800x |
| window=64 | 26x |
| window=1k | 180x |

## 2. Tasks are tiny: 32 instructions on average

## 3. Independent tasks are far away in program order



Can execute *N* tasks ahead of the earliest active task

**Need a large window of speculation**

# Prior Work Can’t Mine Ordered Parallelism

# Prior Work Can’t Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

# Prior Work Can't Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

| Max parallelism | TLS parallelism |
|---|---|
| 800x | 1.1x |

# Prior Work Can’t Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

| Max parallelism | TLS parallelism |
|---|---|
| 800x | 1.1x |

Execution order ≠ creation order

# Prior Work Can't Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

| Max parallelism | TLS parallelism |
|---|---|
| 800x | 1.1x |

Execution order ≠ creation order

Task-scheduling priority queues introduce false data dependences

# Prior Work Can't Mine Ordered Parallelism

- □ Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

| Max parallelism | TLS parallelism |
|---|---|
| 800x | 1.1x |

Execution order ≠ creation order

Task-scheduling priority queues introduce false data dependences

- □ Sophisticated parallel algorithms yield limited speedup

# Prior Work Can't Mine Ordered Parallelism

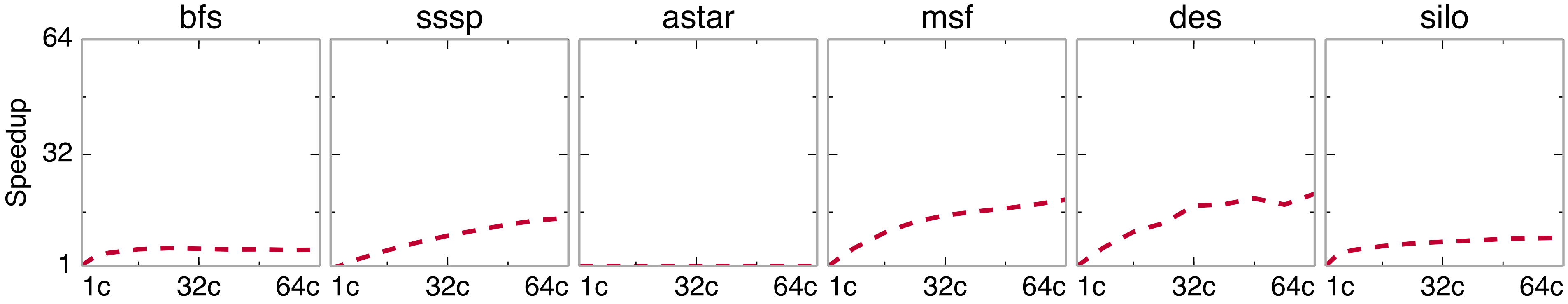
- □ Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

| Max parallelism | TLS parallelism |
|---|---|
| 800x | 1.1x |

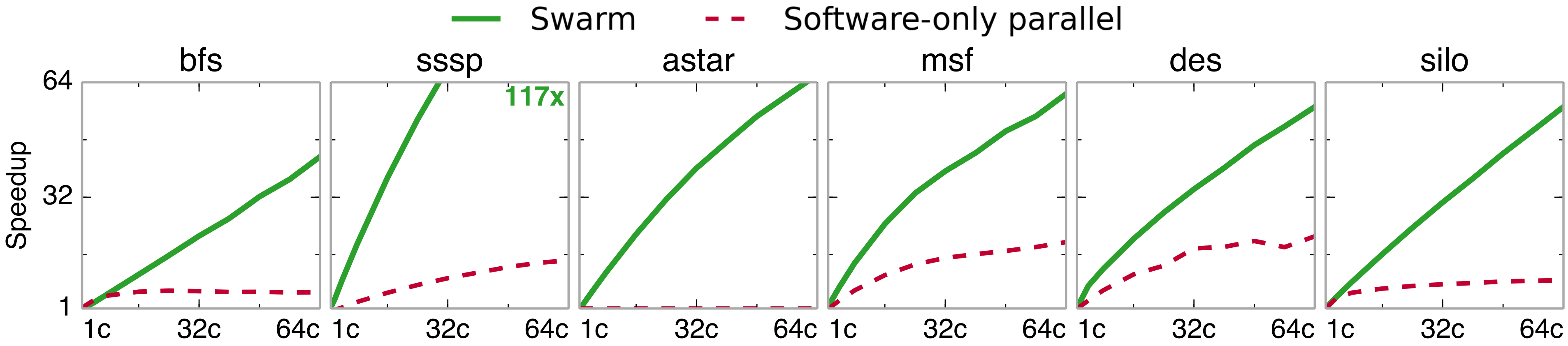
Execution order ≠ creation order

Task-scheduling priority queues introduce false data dependences

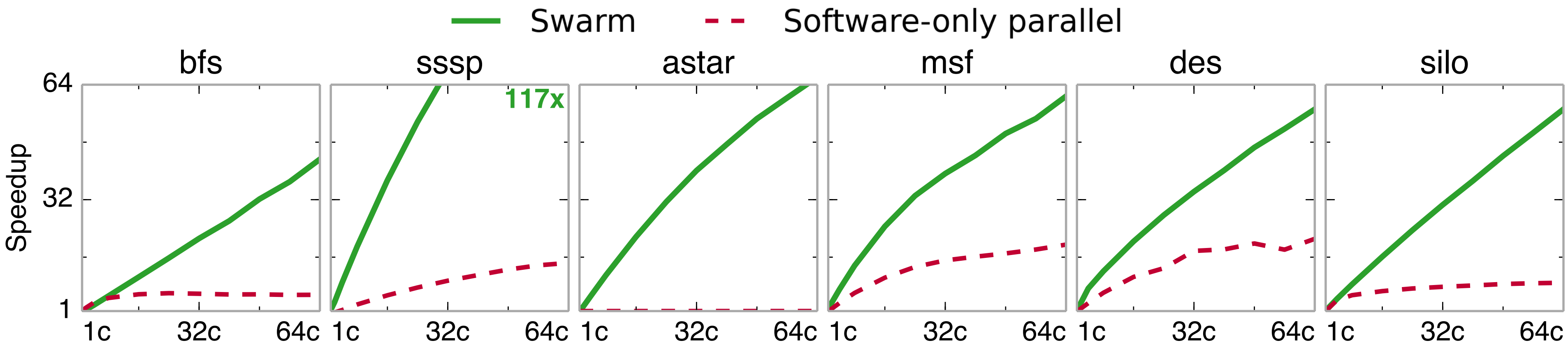
- □ Sophisticated parallel algorithms yield limited speedup



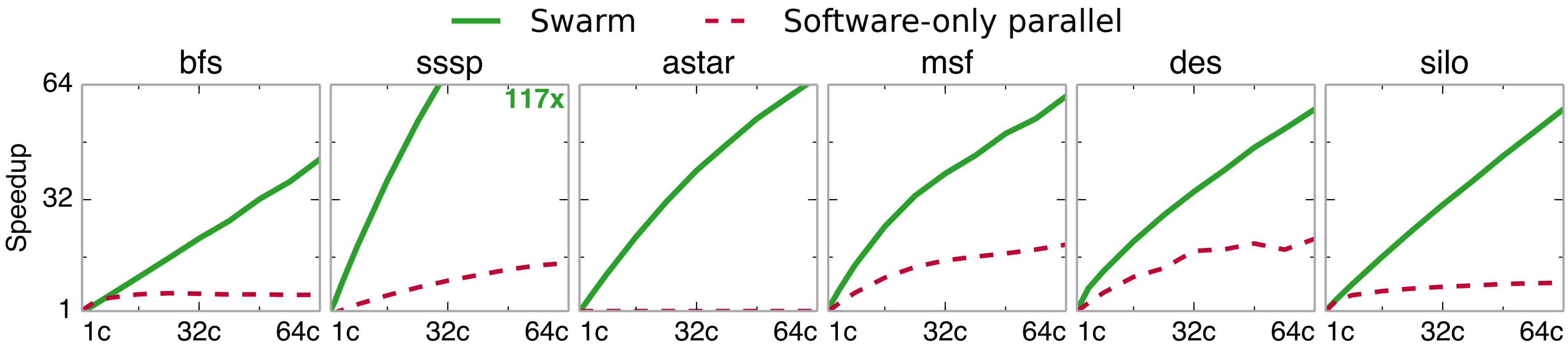
# Swarm Mines Ordered Parallelism



# Swarm Mines Ordered Parallelism

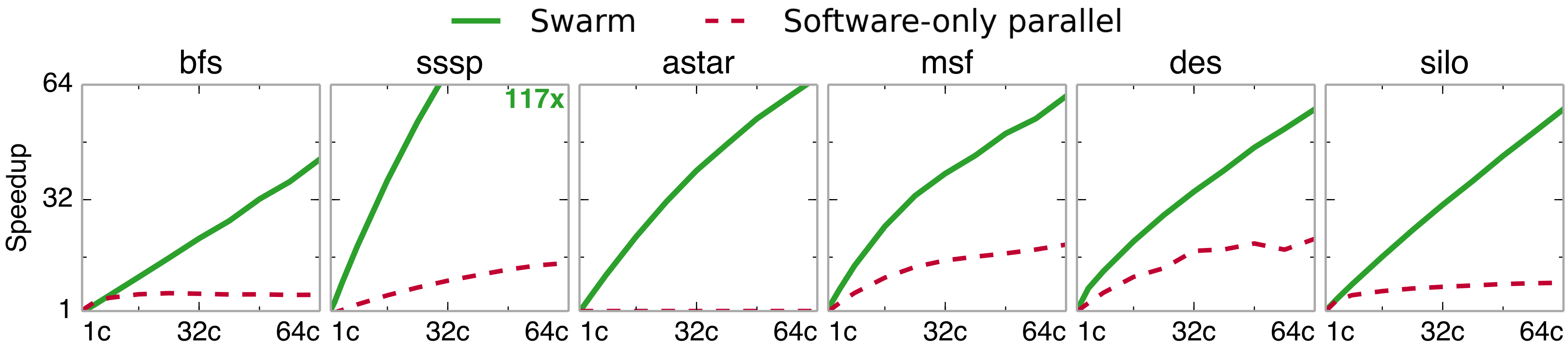


# Swarm Mines Ordered Parallelism



□ Execution model based on timestamped tasks

# Swarm Mines Ordered Parallelism



- Execution model based on timestamped tasks
- Architecture executes tasks speculatively out of order
  - Leverages execution model to scale

# Outline

- □ Understanding Ordered Parallelism
- □ Swarm
- □ Evaluation

# Swarm Execution Model

Programs consist of timestamped tasks

# Swarm Execution Model

Programs consist of timestamped tasks

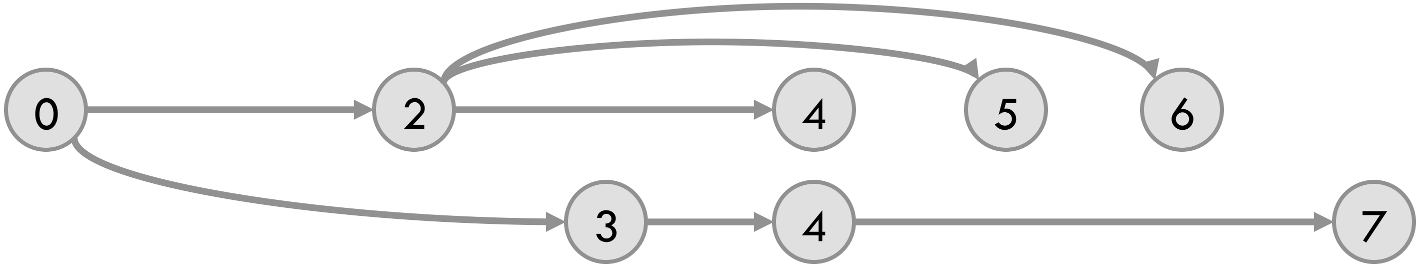
- Tasks can create children tasks with >= timestamp
- Tasks appear to execute in timestamp order

# Swarm Execution Model

Programs consist of timestamped tasks

- Tasks can create children tasks with >= timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

```
swarm::enqueue(fptr, ts, args...);
```

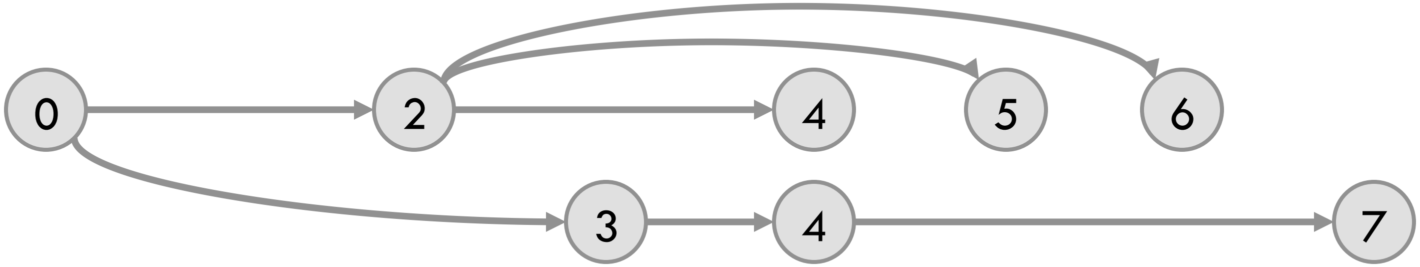


# Swarm Execution Model

Programs consist of timestamped tasks

- Tasks can create children tasks with >= timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

```
swarm::enqueue(fptr, ts, args...);
```



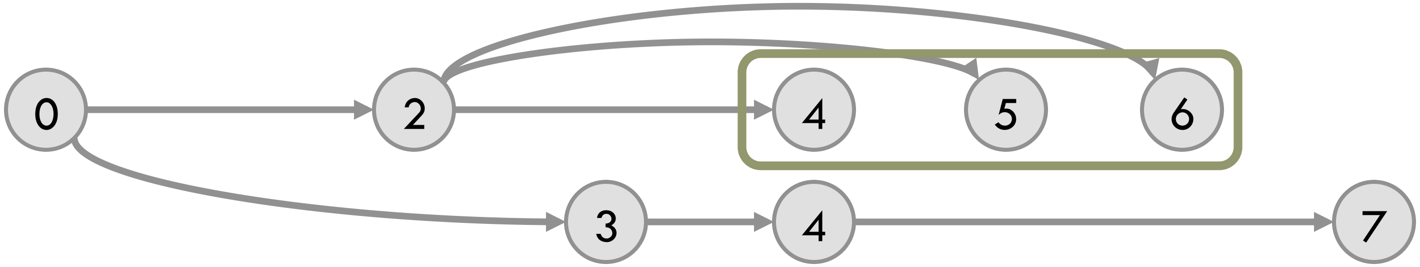
Conveys new work to hardware as soon as possible

# Swarm Execution Model

Programs consist of timestamped tasks

- Tasks can create children tasks with >= timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

```
swarm::enqueue(fptr, ts, args...);
```



Conveys new work to hardware as soon as possible

# Swarm Task Example: Dijkstra

```
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
       Timestamp uDist = dist + edgeWeight(v, u);
       swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}
```

# Swarm Task Example: Dijkstra

```
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
       Timestamp uDist = dist + edgeWeight(v, u);
       swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}
```

# Swarm Task Example: Dijkstra

```
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
      Timestamp uDist = dist + edgeWeight(v, u);
      swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}
```

**Timestamp**

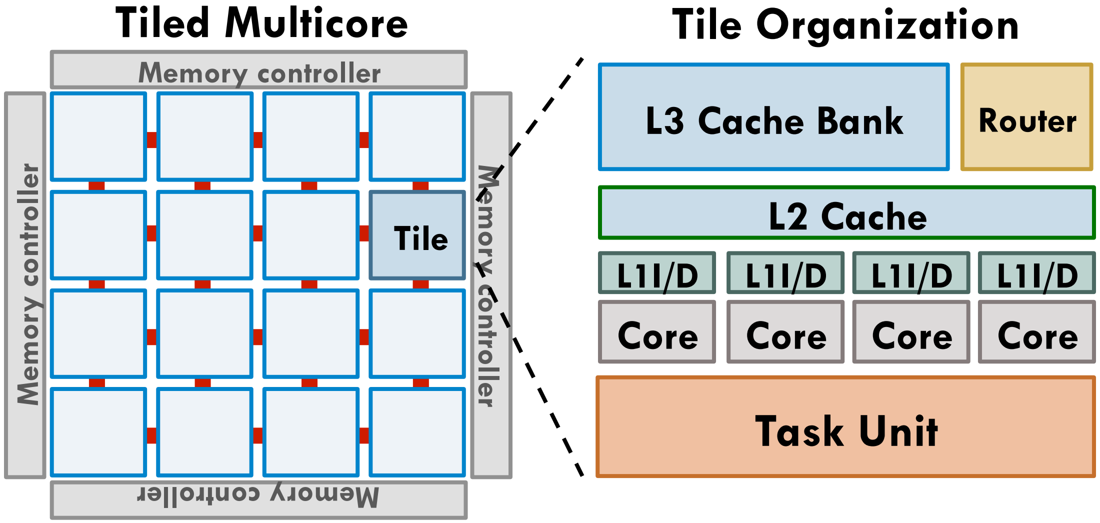
# Swarm Task Example: Dijkstra

```
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
      Timestamp uDist = dist + edgeWeight(v, u);
      swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}
```

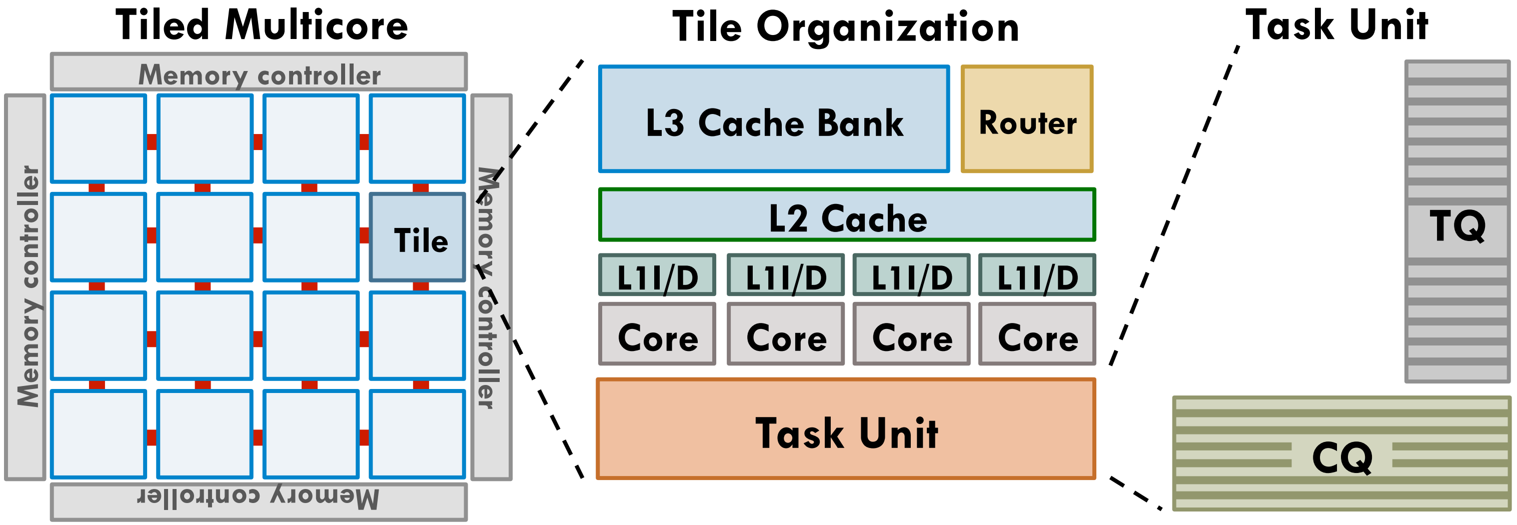
**Timestamp**

```
swarm::enqueue(ssspTask, 0, sourceVertex);
swarm::run();
```

# Swarm Architecture Overview



# Swarm Architecture Overview

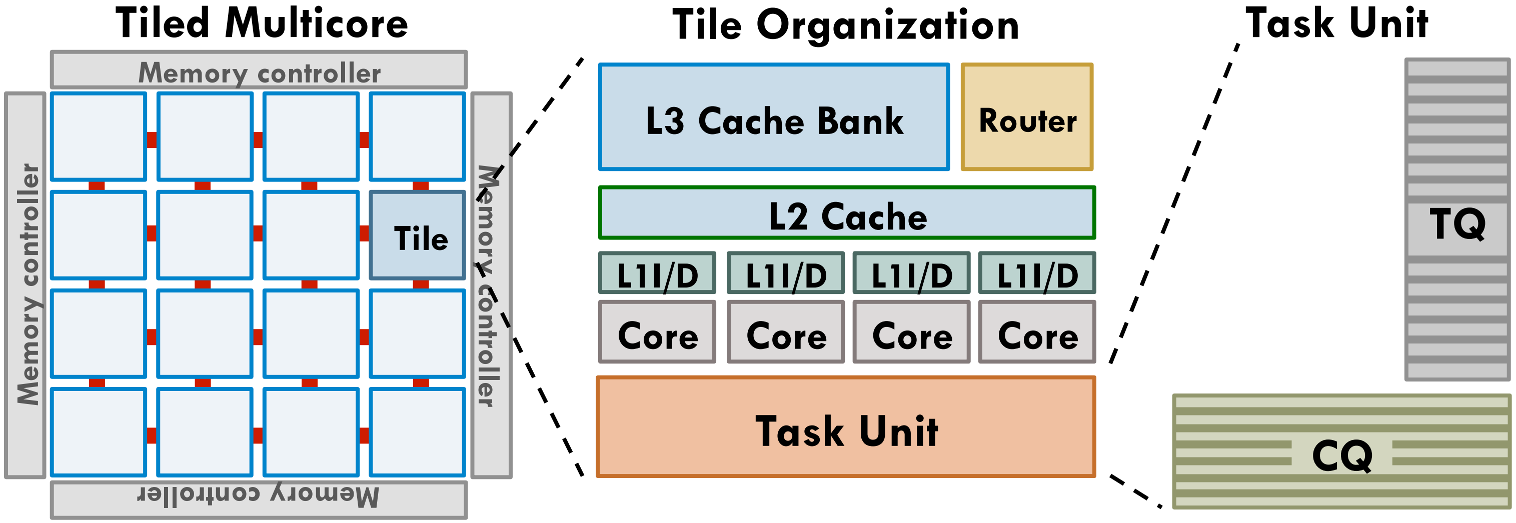


Per-tile task units:

□ **Task Queue:** holds task descriptors

□ **Commit Queue:** holds speculative state of finished tasks

# Swarm Architecture Overview



Per-tile task units:

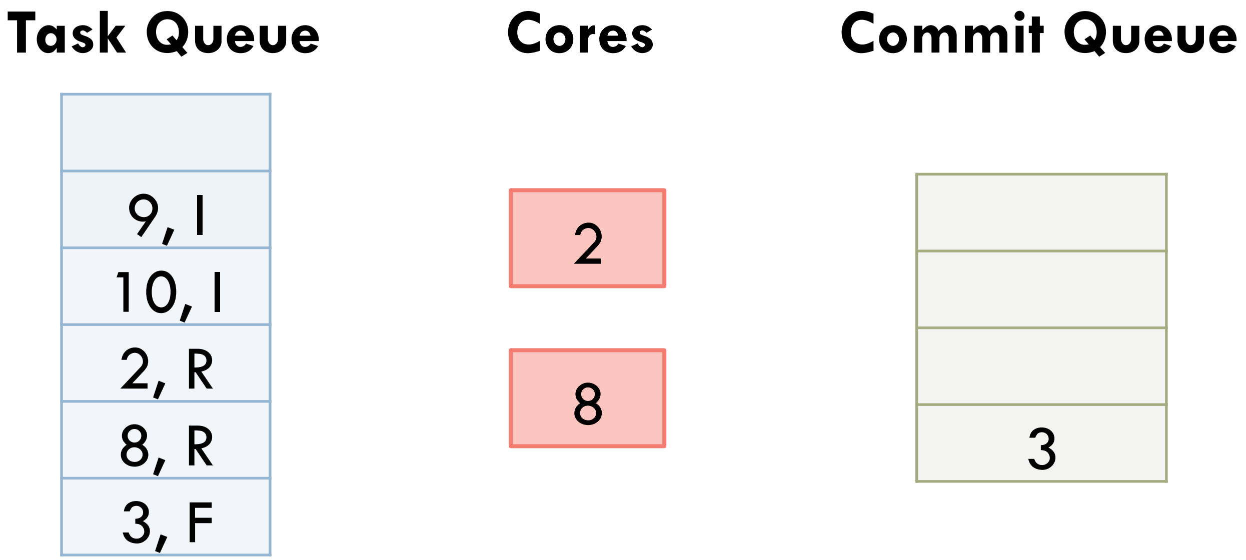
- □ **Task Queue:** holds task descriptors
- □ **Commit Queue:** holds speculative state of finished tasks

Commit queues provide the window of speculation

# Task Unit Queues

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**Task Queue**

9, I
10, I
2, R
8, R
3, F

**Cores**

2
8

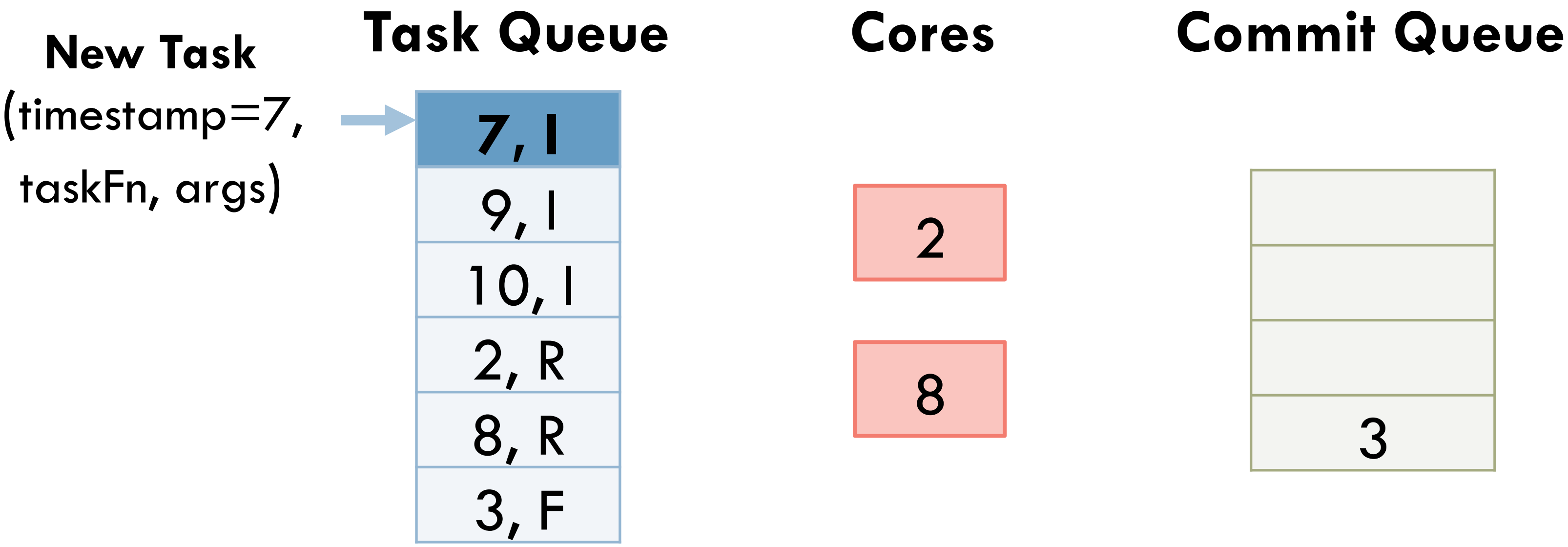
**Commit Queue**

3

# Task Unit Queues

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**New Task**
(timestamp=7, taskFn, args)

**Task Queue**

**7, I**
9, I
10, I
2, R
8, R
3, F

**Cores**

2
8

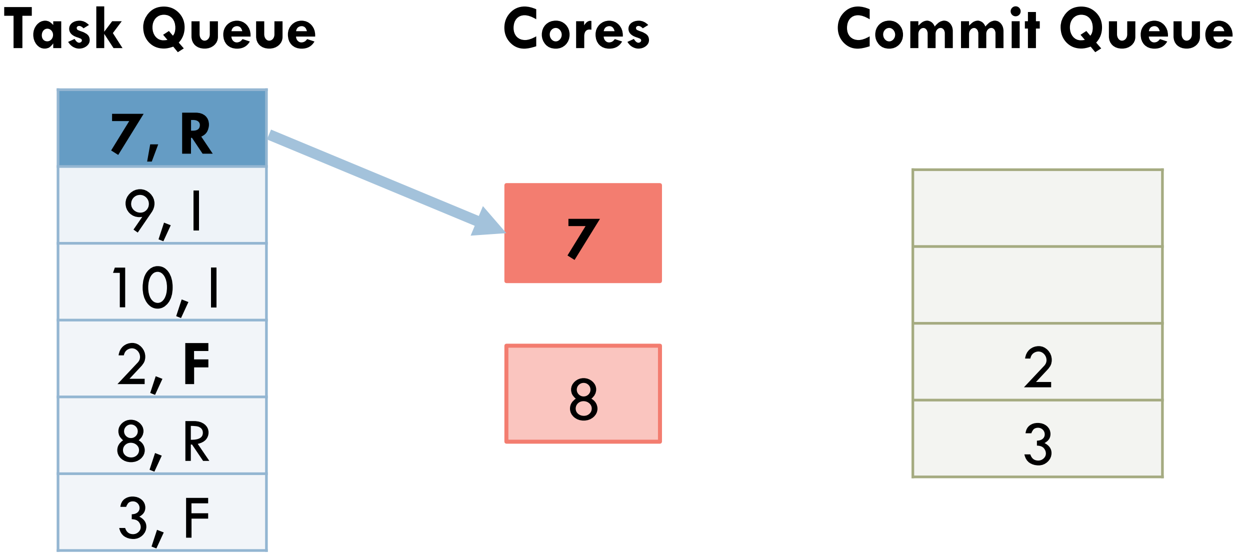
**Commit Queue**

3

# Task Unit Queues

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**Task Queue**

**7, R**
9, I
10, I
2, **F**
8, R
3, F

**Cores**

**7**
8

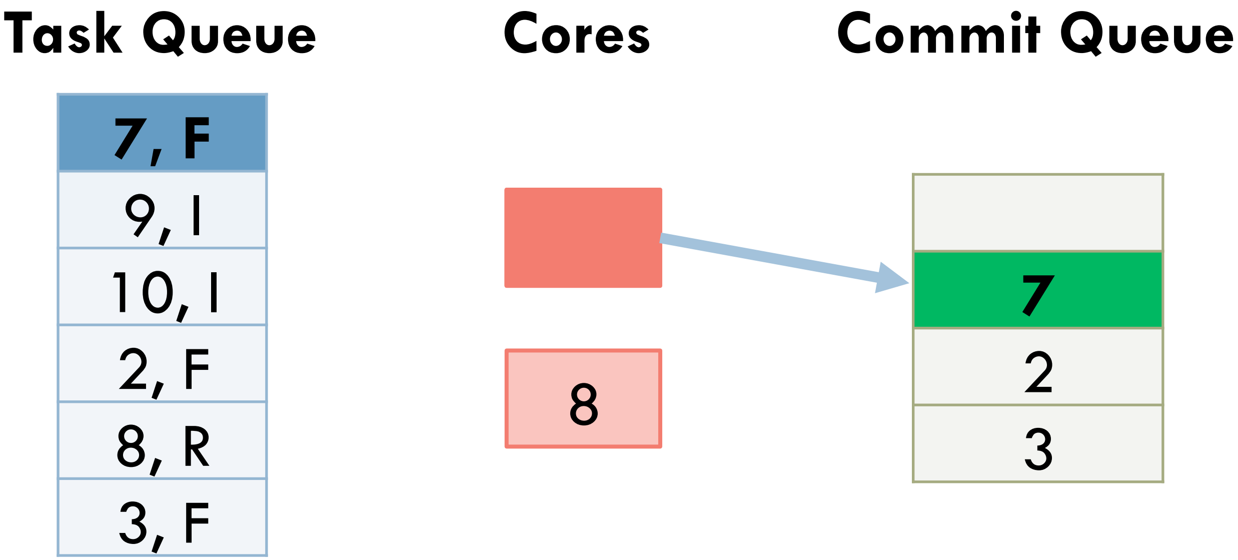
**Commit Queue**

2
3

# Task Unit Queues

- □ **Task queue:** holds task descriptors
- □ **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**Task Queue**

**7, F**
9, I
10, I
2, F
8, R
3, F

**Cores**

8

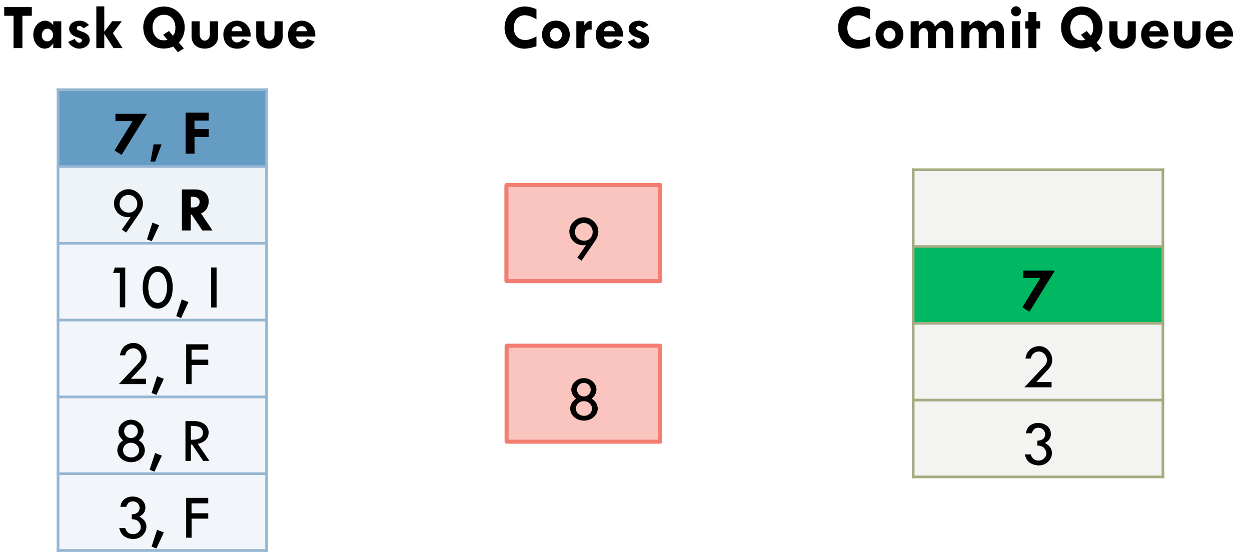
**Commit Queue**

**7**
2
3

# Task Unit Queues

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**Task Queue**

**7, F**
9, **R**
10, I
2, F
8, R
3, F

**Cores**

9
8

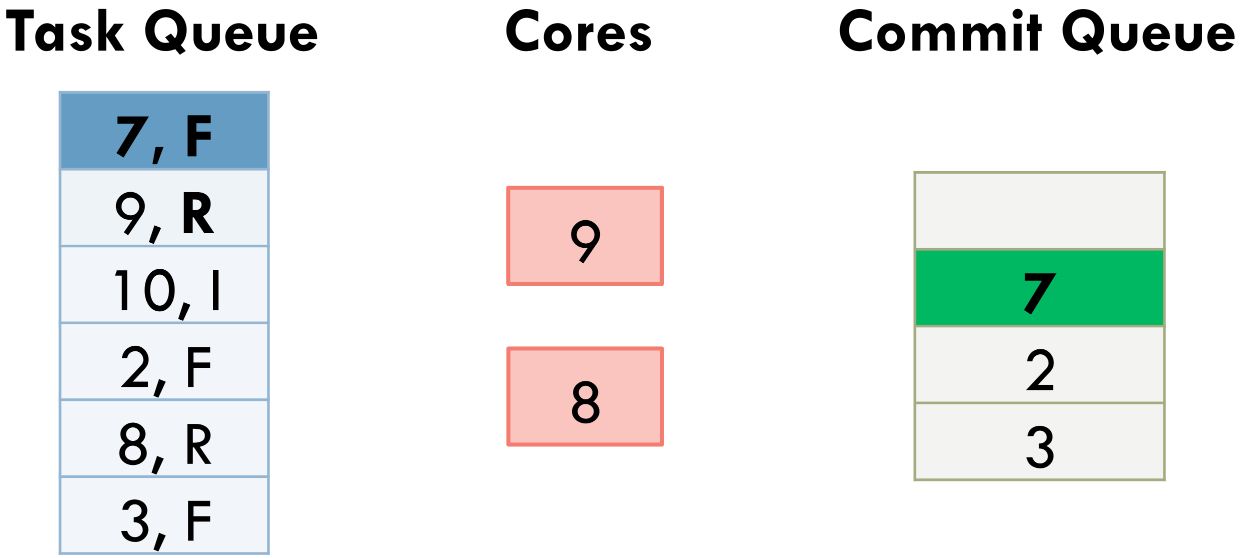
**Commit Queue**

**7**
2
3

# Task Unit Queues

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

**Task States: IDLE (I) RUNNING (R) FINISHED (F)**



**Task Queue**

**7, F**
9, **R**
10, I
2, F
8, R
3, F

**Cores**

9
8

**Commit Queue**

**7**
2
3

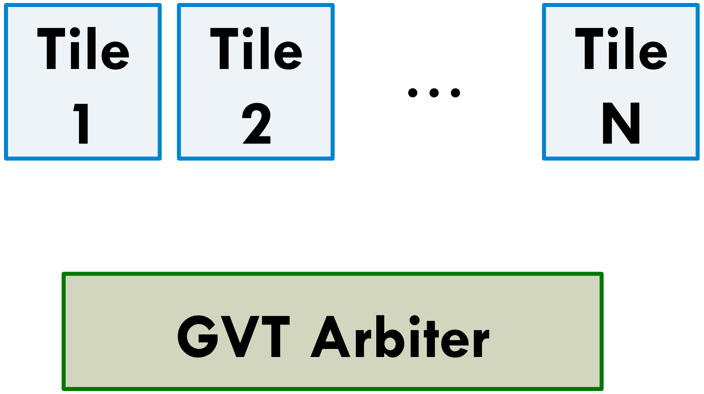
**Similar to a reorder buffer, but at the task level**

# High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow

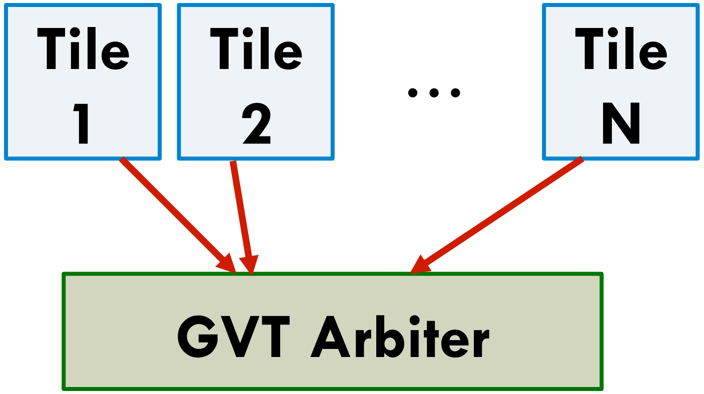
# High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]



# High-Throughput Ordered Commits

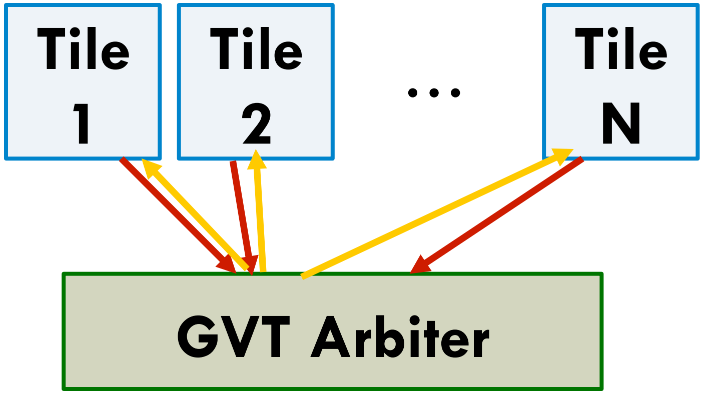
- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]



- Tiles periodically communicate to find the earliest unfinished task

# High-Throughput Ordered Commits

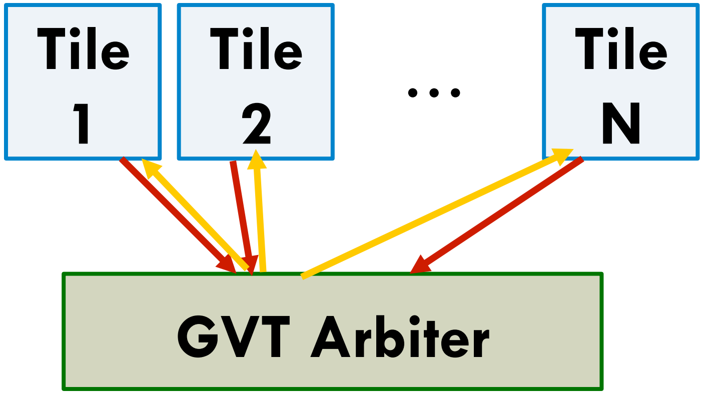
- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]



- Tiles periodically communicate to find the earliest unfinished task

# High-Throughput Ordered Commits

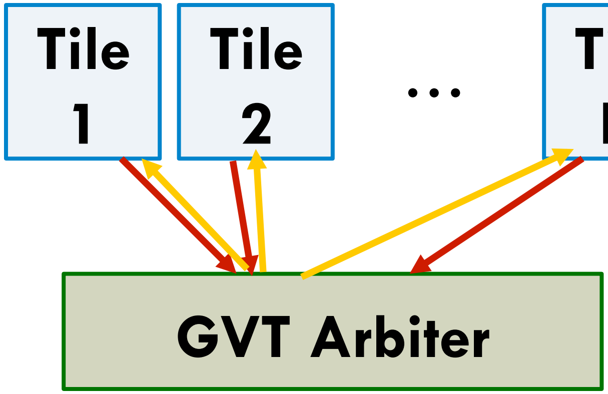
- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]



- Tiles periodically communicate to find the earliest unfinished task
- Tiles commit all tasks that precede it

# High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]



Tile 1

Tile 2

…

Tile N

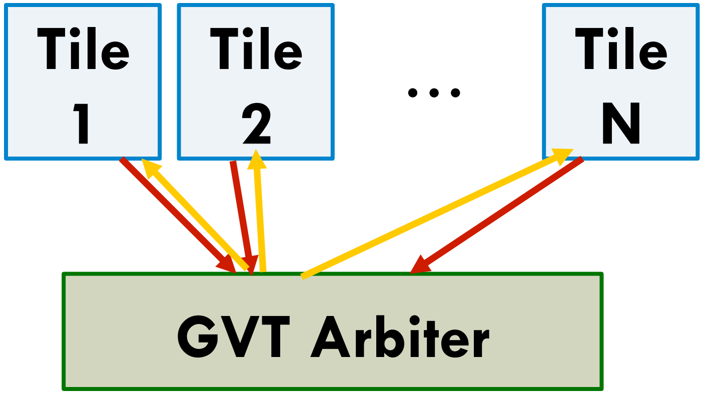
GVT Arbiter

- Tiles periodically communicate to find the earliest unfinished task
- Tiles commit all tasks that precede it

With large commit queues, many tasks commit at once

# High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - **1 task commit/cycle** to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt "Virtual Time" [Jefferson, TOPLAS 1985]

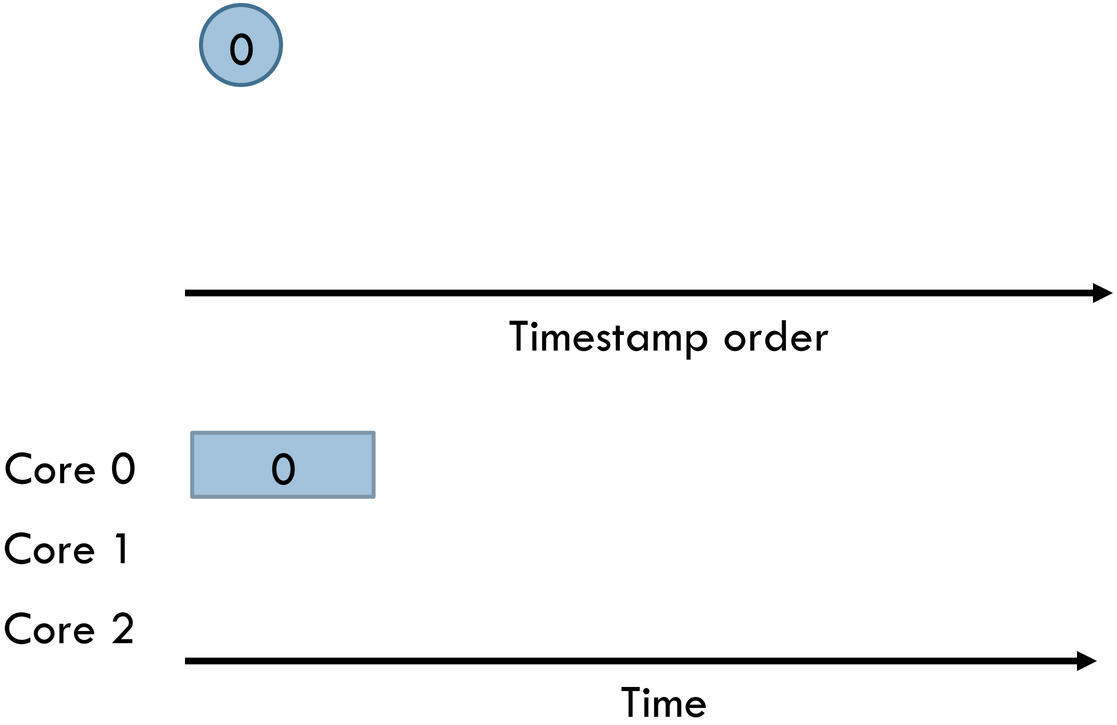


- Tiles periodically communicate to find the earliest unfinished task
- Tiles commit all tasks that precede it

With large commit queues, many tasks commit at once

Amortizes commit costs among many tasks

# Speculative Execution Example



0

Timestamp order

Core 0

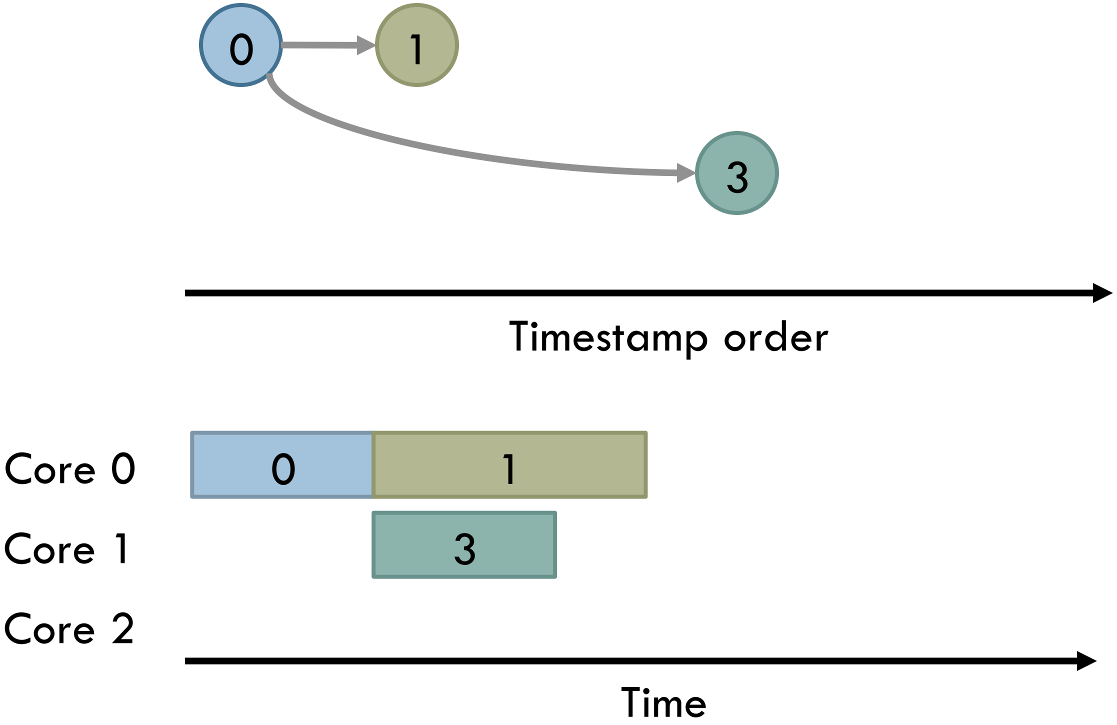
Core 1

Core 2

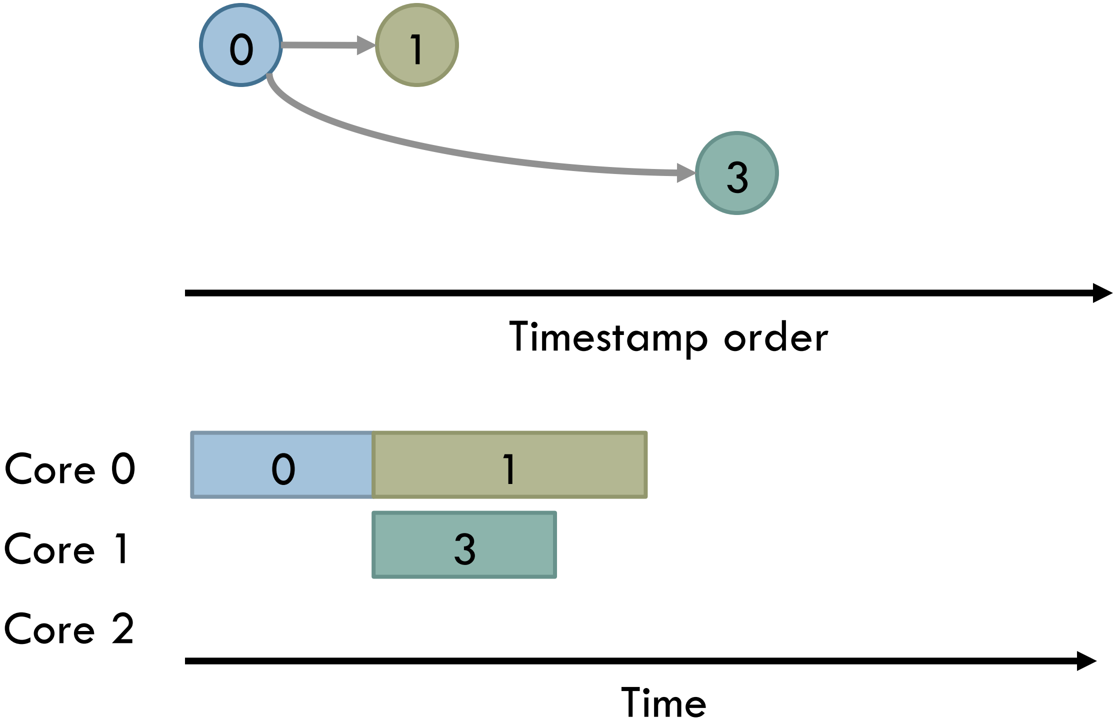
0

Time

# Speculative Execution Example

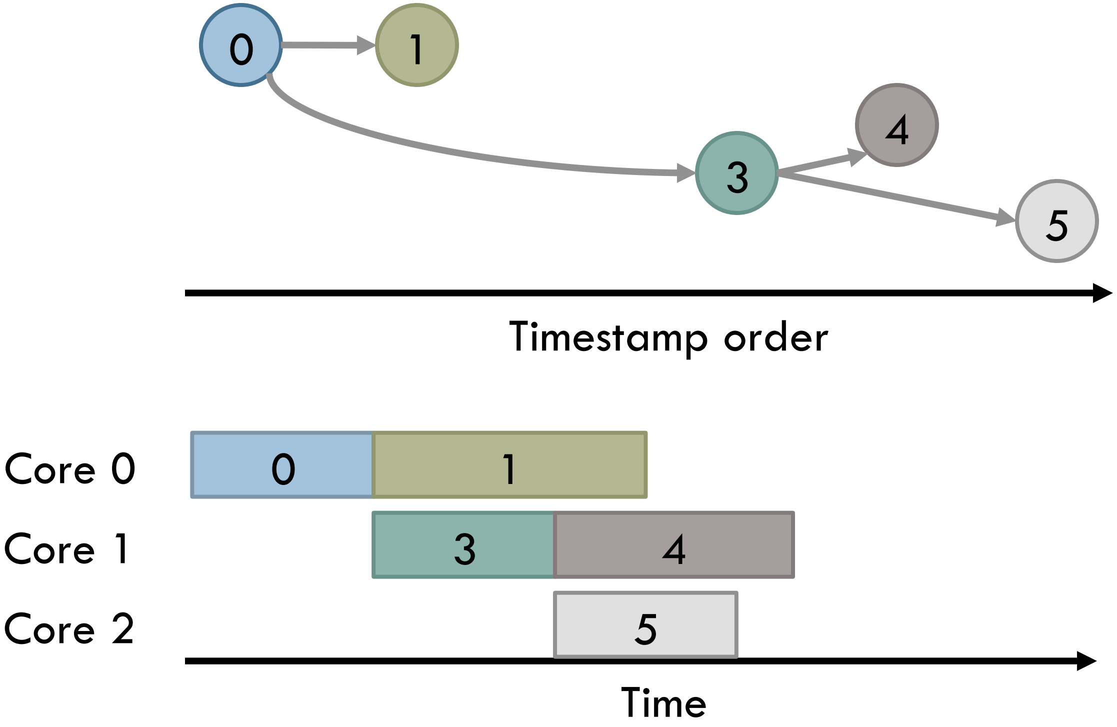


# Speculative Execution Example



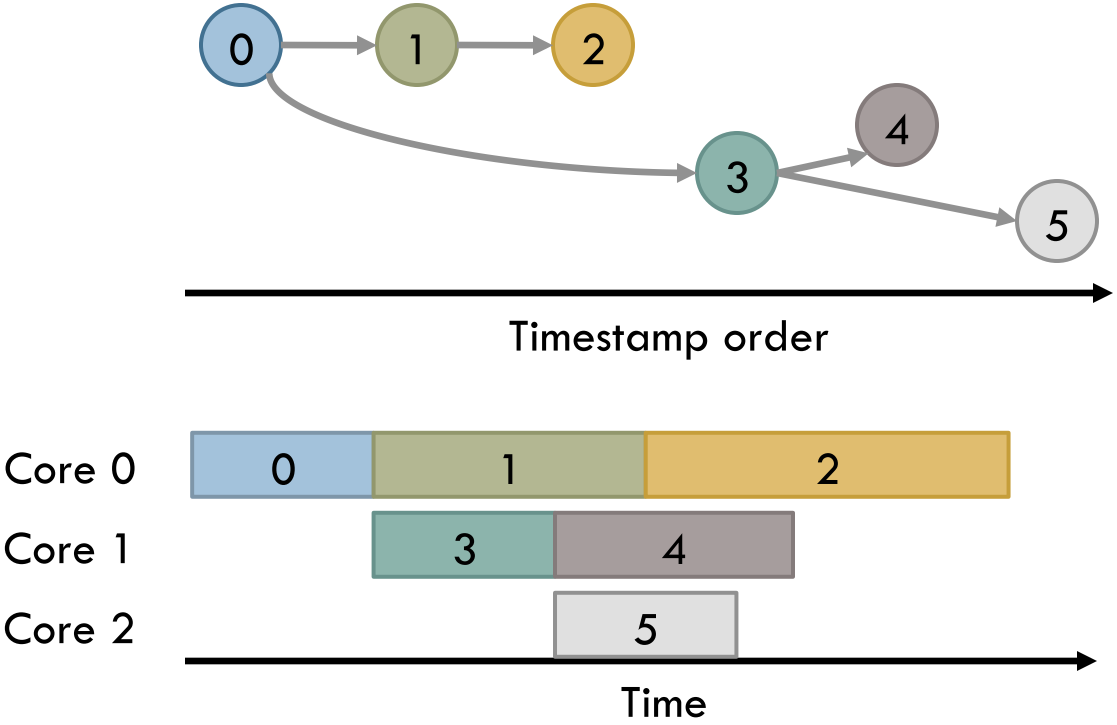
- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism

# Speculative Execution Example



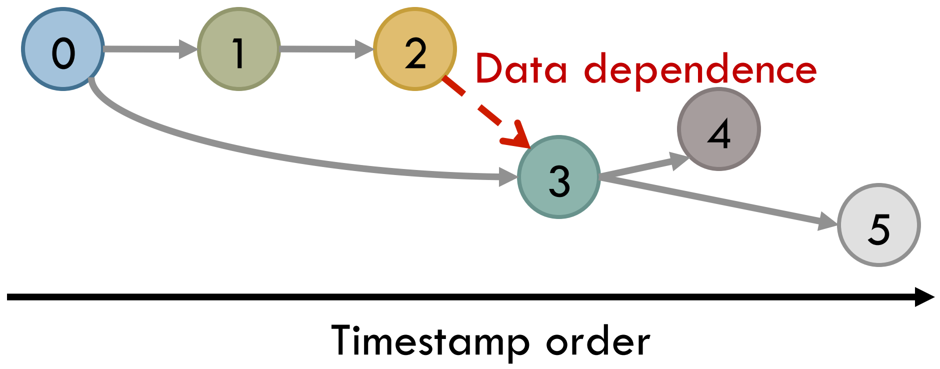
- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism

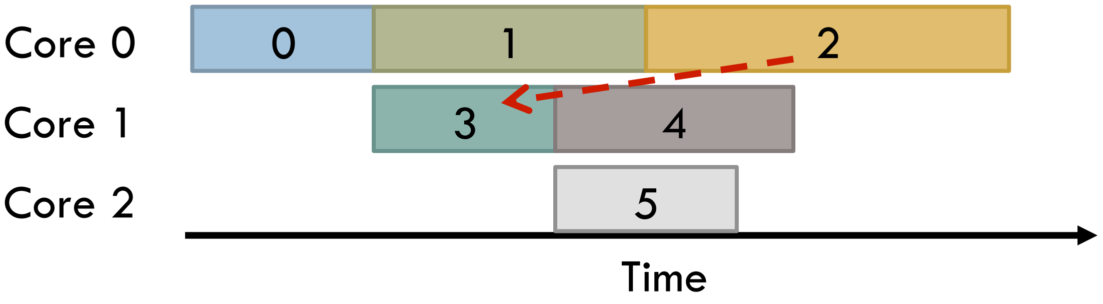
# Speculative Execution Example



- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism

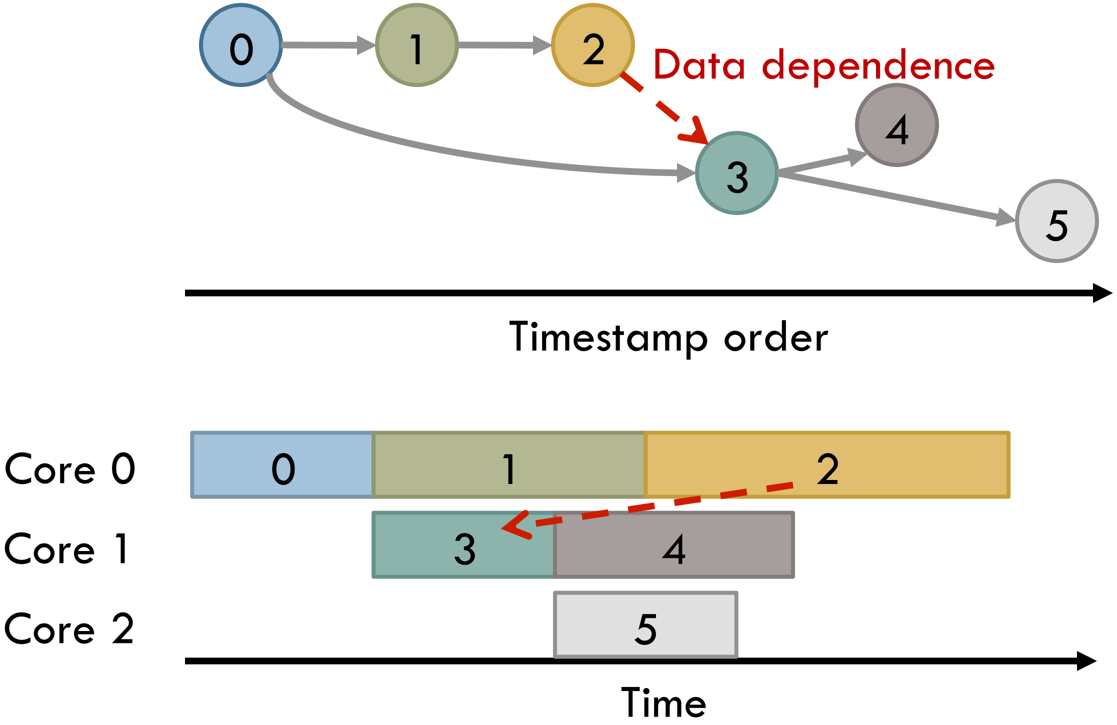
# Speculative Execution Example





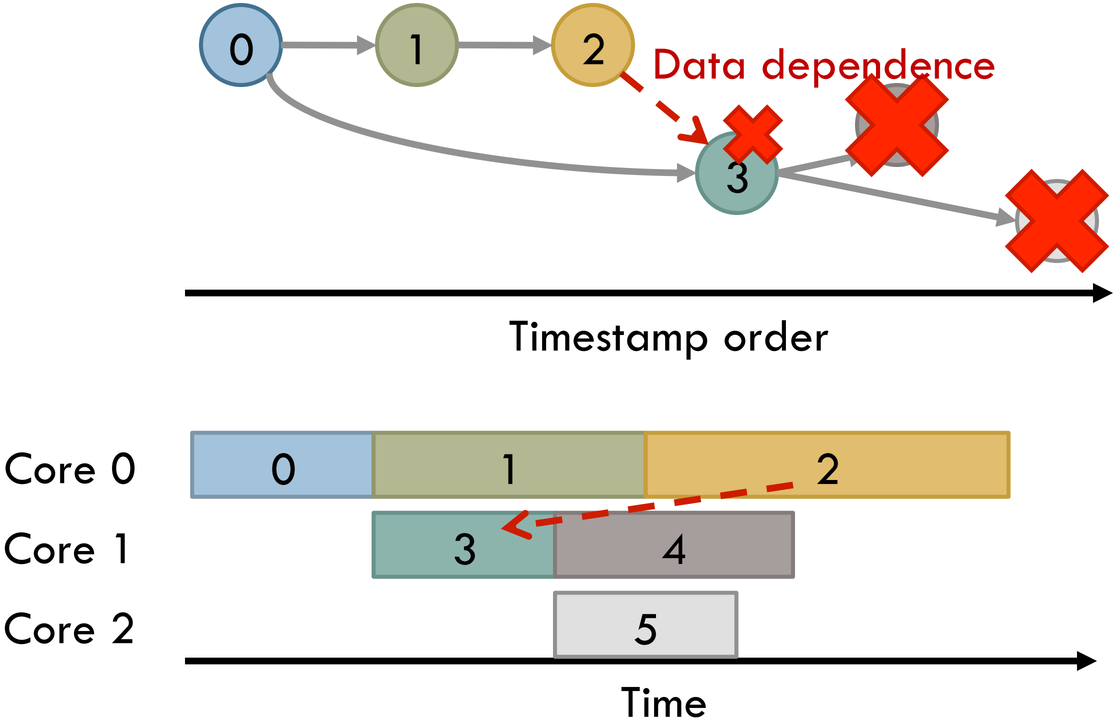
- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism

# Speculative Execution Example



- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism
  - May trigger cascading (but selective) aborts

# Speculative Execution Example



- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism
  - May trigger cascading (but selective) aborts

# Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state

# Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state
- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]

# Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state

- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]
  - Uses hierarchical memory system to filter conflict checks

# Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state

- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]
  - Uses hierarchical memory system to filter conflict checks

- Enables two helpful properties
  1. **Forwarding** of still-speculative data
  2. On rollback, corrective writes **abort dependent tasks only**

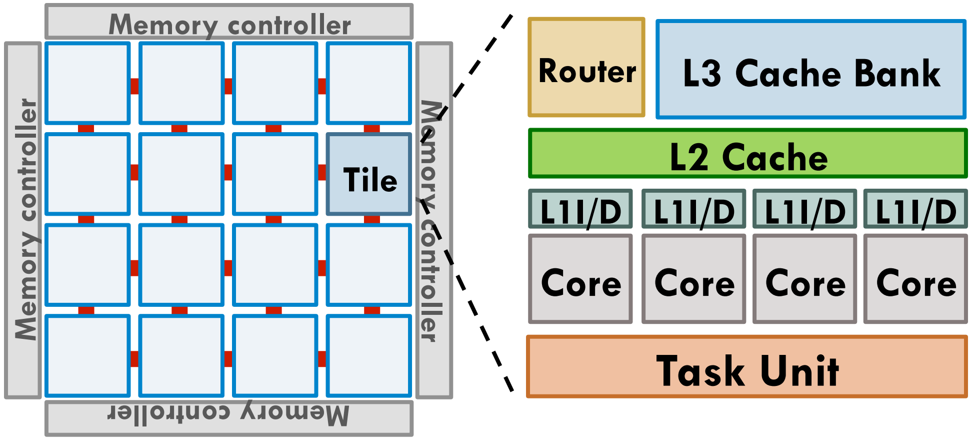
# Outline

- □ Understanding Ordered Parallelism
- □ Swarm
- □ Evaluation

# Evaluation Methodology

- □ Event-driven, sequential, Pin-based simulator

- □ Target system: 64-core, 16-tile chip



16 MB shared L3 (1MB/tile)

256 KB per-tile L2s

32 KB per-core L1s

4096 task queue entries (64/core)

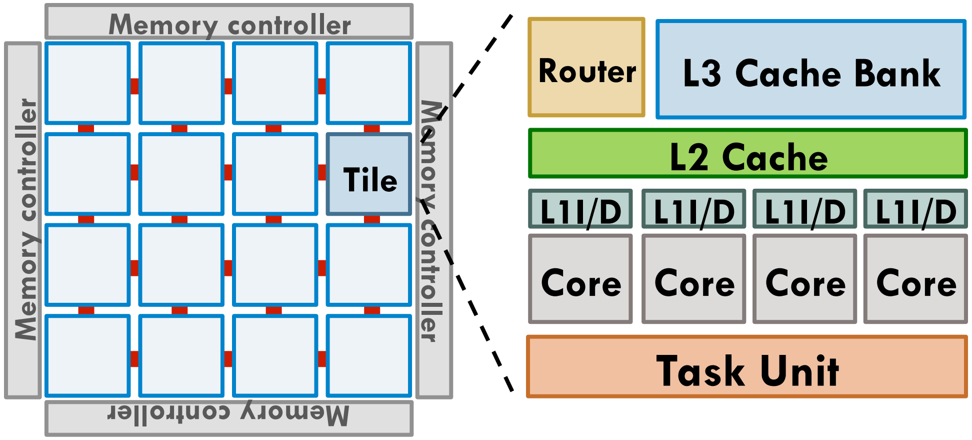
1024 commit queue entries (16/core)

256-byte, 8-way Bloom filters

# Evaluation Methodology

- □ Event-driven, sequential, Pin-based simulator

- □ Target system: 64-core, 16-tile chip



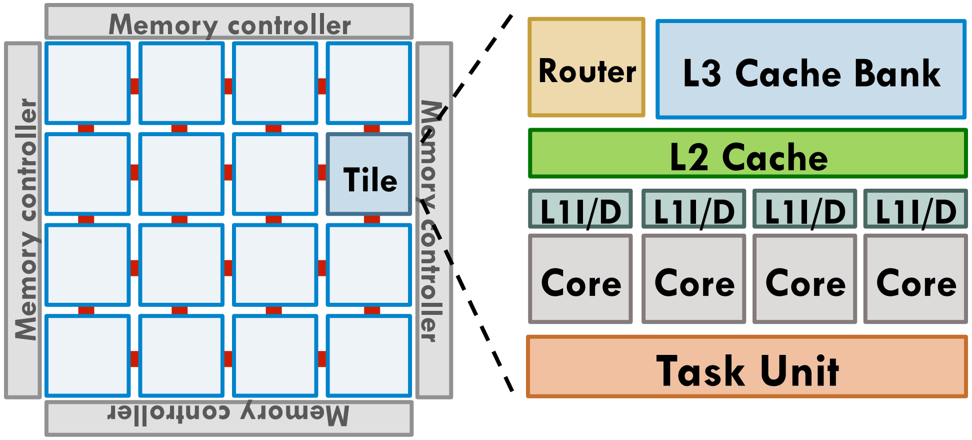
16 MB shared L3 (1MB/tile)
256 KB per-tile L2s
32 KB per-core L1s

4096 task queue entries (64/core)
1024 commit queue entries (16/core)
256-byte, 8-way Bloom filters

# Evaluation Methodology

- Event-driven, sequential, Pin-based simulator

- Target system: 64-core, 16-tile chip

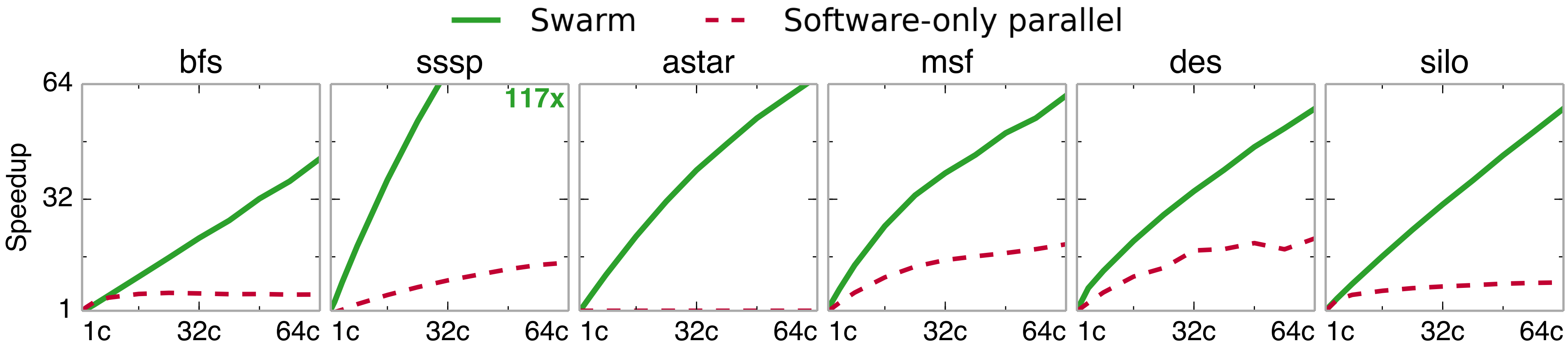


16 MB shared L3 (1MB/tile)
256 KB per-tile L2s
32 KB per-core L1s

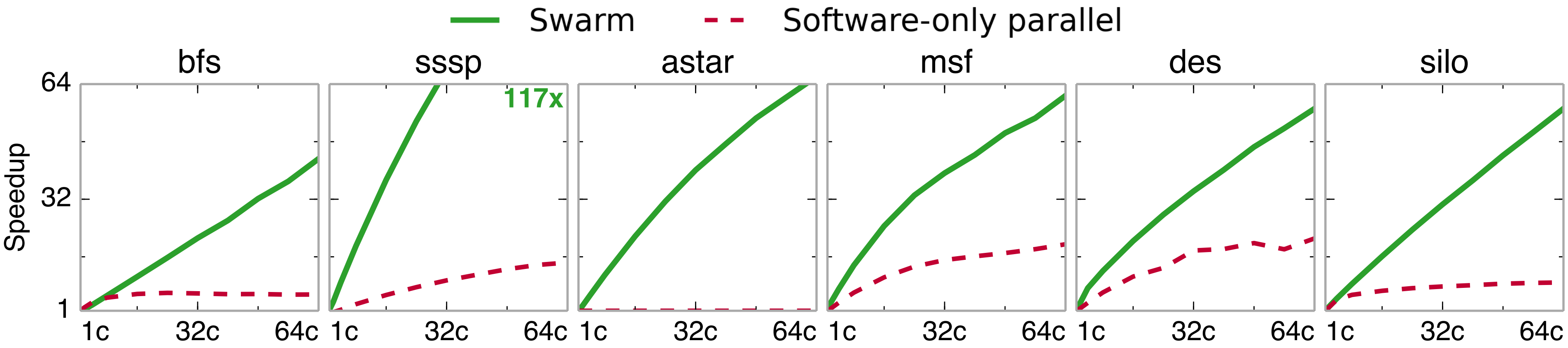
4096 task queue entries (64/core)
1024 commit queue entries (16/core)
256-byte, 8-way Bloom filters

- Scalability experiments from 1-64 cores
  - Scaled-down systems have fewer tiles

# Swarm vs. Software Versions

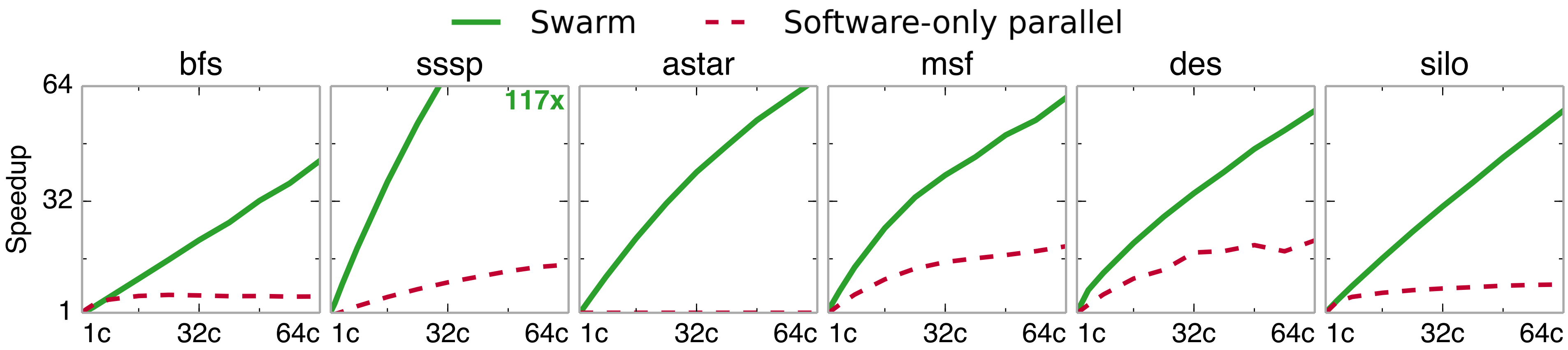


# Swarm vs. Software Versions



**43x – 117x faster than serial versions**

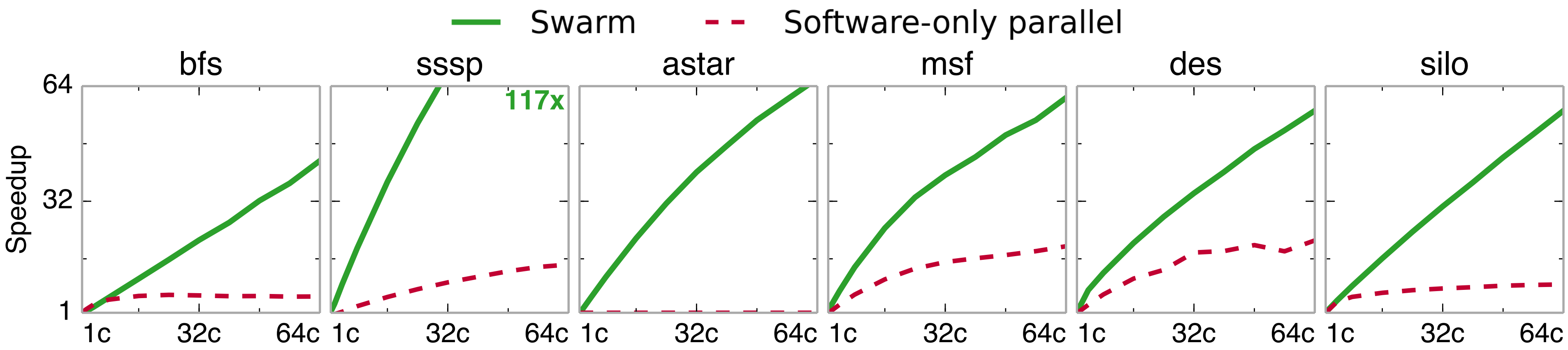
# Swarm vs. Software Versions



**43x – 117x faster than serial versions**

**3x – 18x faster than parallel versions**

# Swarm vs. Software Versions

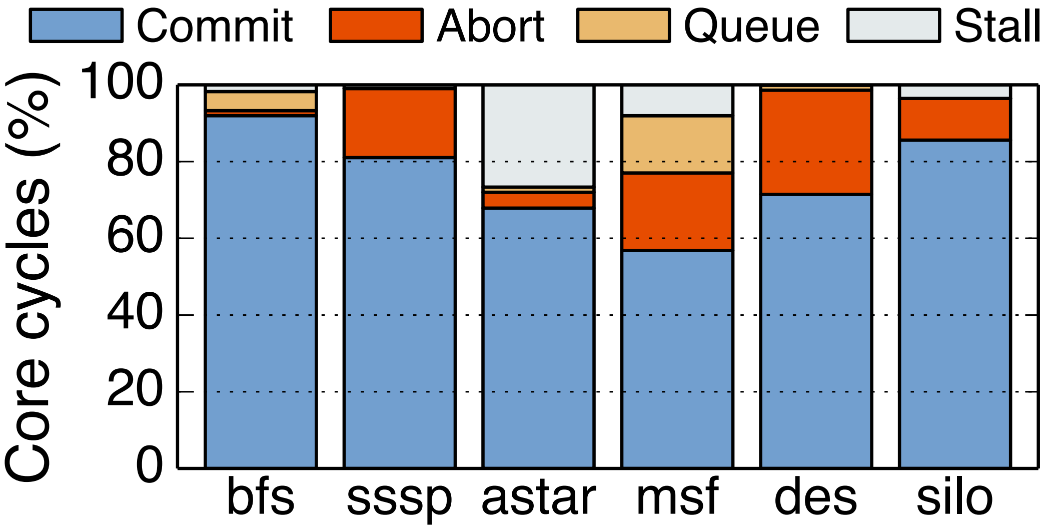


**43x – 117x faster than serial versions**

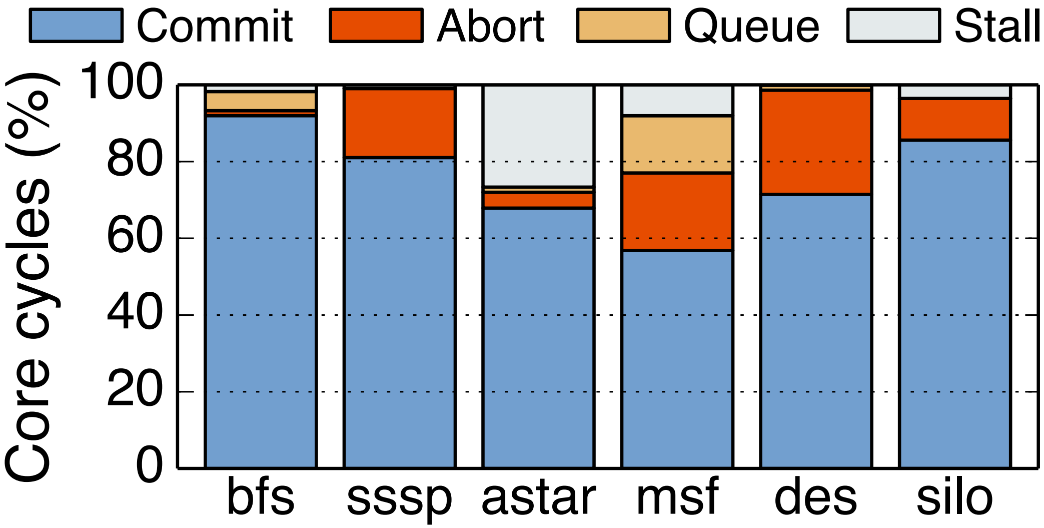
**3x – 18x faster than parallel versions**

**Simple implicitly-parallel code**

# Swarm Uses Resources Efficiently

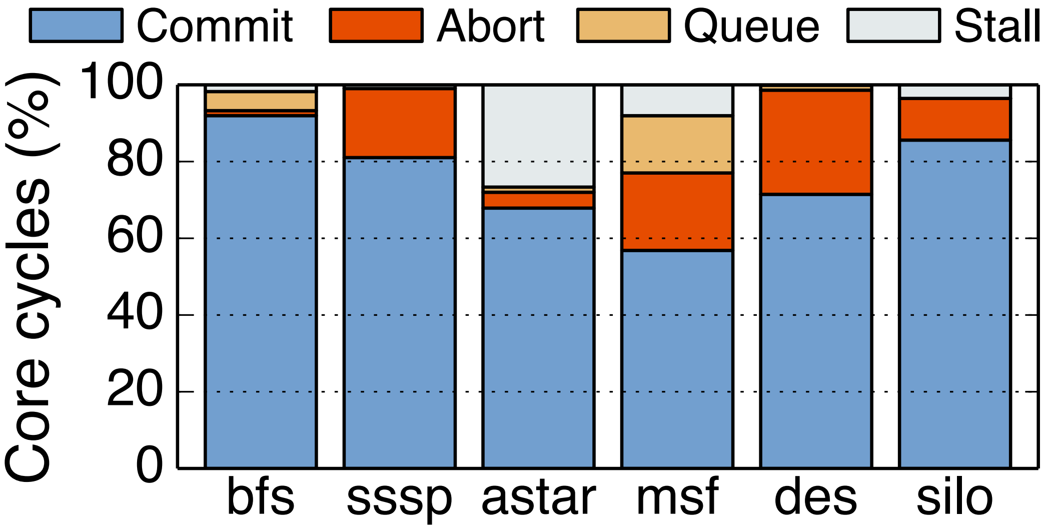


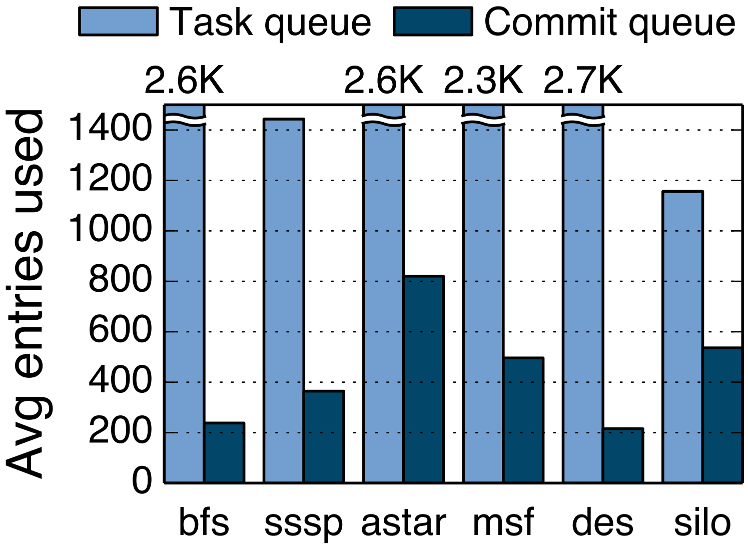
# Swarm Uses Resources Efficiently



**Most time spent executing tasks that commit**

# Swarm Uses Resources Efficiently

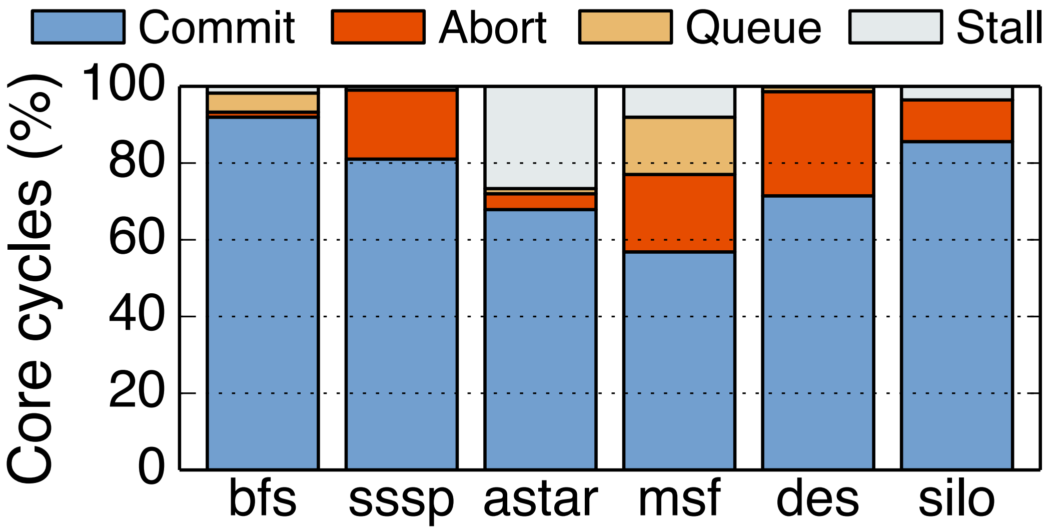


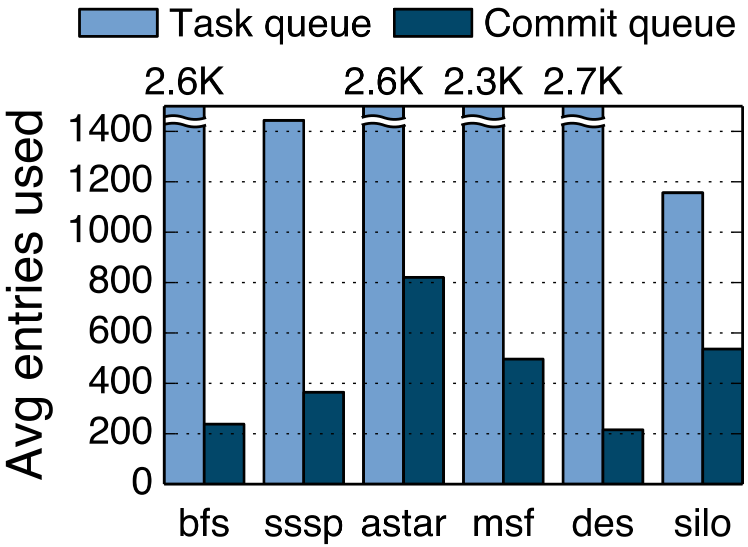


**Most time spent executing tasks that commit**

**Swarm speculates 200-800 tasks ahead on average**

# Swarm Uses Resources Efficiently





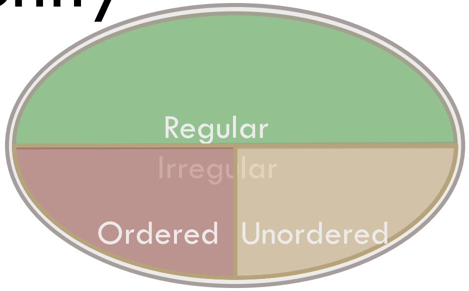
**Most time spent executing tasks that commit**

**Swarm speculates 200-800 tasks ahead on average**

- Speculation adds moderate energy overheads:
  - 15% extra network traffic
  - Conflict check logic triggered in 9-16% of cycles

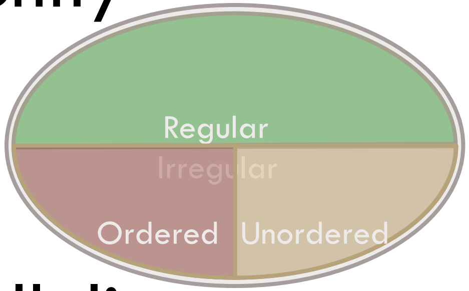
# Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general



# Conclusions

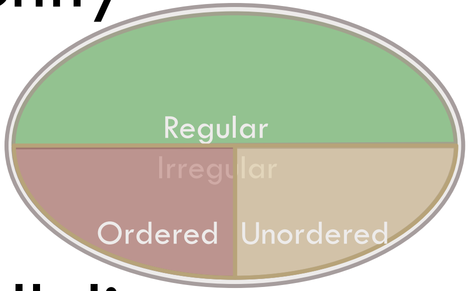
- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general



- Conventional wisdom: Ordering limits parallelism

# Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general

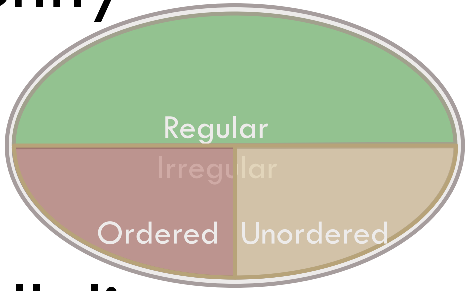


- ~~Conventional wisdom: Ordering limits parallelism~~

Expressive execution model + large window =
Only true data dependences limit parallelism

# Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general



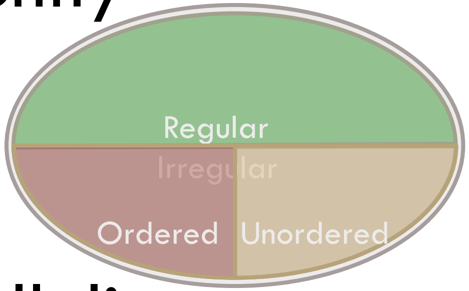
- ~~Conventional wisdom: Ordering limits parallelism~~

Expressive execution model + large window =
Only true data dependences limit parallelism

- Conventional wisdom: Speculation is wasteful

# Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general



- ~~Conventional wisdom: Ordering limits parallelism~~

  Expressive execution model + large window =

  Only true data dependences limit parallelism

- ~~Conventional wisdom: Speculation is wasteful~~

  Speculation unlocks plentiful ordered parallelism

  Can trade parallelism for efficiency (e.g., simpler cores)

# Thanks for your attention!
# Questions?

*A Scalable Architecture for Ordered Parallelism*

Mark Jeffrey, Suvinay Subramanian, Cong Yan,
Joel Emer, Daniel Sanchez

Massachusetts Institute of Technology

CSAIL

NVIDIA